# 四、Spring源码解析

Default开头的类就是默认实现类

spring-core.jar包是最顶层的，所有项目都要依赖

spring-bean.jar包定义的是规范

spring-context.jar包是工厂的实现、DI的实现

spring-aop.jar包是面向切面编程

spring-aspects.jar包是横切的意思

Spring中的单例是使用Map实现的

spring中的对象如果不声明，默认都是单例

对象被代理之后，相当于把类名都改了

spring IOC在初始化时（定位、加载、注册），将Bean的描述BeanDefinition存储在Spring IOC容器中，存储的数据类型是ConcurrentHashMap，key为Bean的name，value为BeanDefinition。 Spring IOC容器中还有一个别名的容器，也是ConcurrentHashMap进行保存的

Spring IOC的依赖注入时（第一步、读取BeanDefinition获取起来关系转换成BeanWrapper。第二步、createBeanInstance方法实例化（代理对象）。第三步、注入：设置），将实例化的Bean也都存放在Spring IOC容器中，存储的数据类型是ConcurrentHashMap，key为Bean的name，value为Object类。（Spring IOC有一个lazy-init设置，如果它为true，则在初始化的时候就会创建所有对象的实例，延时加载是针对getBean的动作而言的）

IOC用到了什么模式，用在了哪里：

工厂模式：

代理模式：

单例模式：

委派模式：

原型模式：

模板模式：

## 1、什么是IOC/DI

IOC（Inversion of Control）控制反转：所谓控制反转，就是把原先我们的代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一个描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们可配置的文件

DI（Dependency Injection）依赖注入：就是指对象是被动接受依赖类而不是自己主动去找，换句话说就是指对象不是从容器中查找它依赖的类，而是在容器实例化对象的时候主动将它的类注入给它

## 2、Spring IOC体系结构

### （1）BeanFactory

SpringBean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用。

其中BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范，BeanFactory有三个子类：ListableBeanFactory、HierarchicalBeanFactory和AutowireCapableBeanFactory。但是最终实现类是DefaultListableBeanFactory，他实现了所有接口。那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在Spring内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如：ListableBeanFactory接口表示这些Bean是可列表的，而HierarchiacaBeanFactory表示的是这些Bean有继承关系的，也就是每个Bean有可能有父Bean。AutowireCapableBeanFactory接口定义Bean的自动装配规则。这四个接口共同定义了Bean的集合。Bean之间的关系、以及Bean行为

BeanFactory源码：

public interface BeanFactory {

String FACTORY\_BEAN\_PREFIX = "&";

/\* 根据bean的名字，获取在IOC容器中得到bean实例 \*/

Object getBean(String name) throws BeansException;

/\* 根据bean的名字和class类型来得到Bean实例，增加类型安全验证机制 \*/

<T> T getBean(String name, Class<T> requiredType) throws BeansException;

<T> T getBean(Class<T> requiredType) throws BeansException;

Object getBean(String name, Object... args) throws BeansException;

boolean containsBean(String name);

boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

boolean isPrototype(String name) throws NoSuchBeanDefinitionException;

boolean isTypeMatch(String name, Class<?> targetType) throws NoSuchBeanDefinitionException;

Class<?> getType(String name) throws NoSuchBeanDefinitionException;

/\* 获取bean的别名 \*/

String[] getAliases(String name);

}

在BeanFactory里只对IOC容器的基本行为做了定义，根本不关系你的bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

而要知道工厂是如何产生对象的，我们需要看具体的IOC容器实现，Spring提供了许多IOC容器的实现，比如XmlBeanFactory，ClasspathXmlApplicationContext等。其中XmlBeanFactory就是针对最基本的IOC容器的实现，这个IOC容器可以读取XML文件定义的BeanDefinition（Xml文件中对bean的描述），如果说XmlBeanFactory是容器中的屌丝，ApplicationContext应该算是容器中的高富帅

ApplicationContext是Spring提供的一个高级IOC容器，它除了能够提供IOC容器的基本功能外，还为用户提供了以下的附加服务。

从ApplicationContext接口实现，我们看出其特点：

1. 支持信息源，可以实现国际化（实现MessageSource接口）
2. 访问资源。（实现ResourcePatternResolver接口）
3. 支持应用事件。（实现ApplicationEventPublisher接口）

### BeanDefinition

SpringIOC容器管理了我们定义的各种Bean对象及其相互关系，Bean对象在Spring实现中是以BeanDefinition来描述的。

Bean的解析过程非常复杂，功能被分的很细，应为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean的解析主要就是对Spring配置文件的解析。

## 3、IOC容器的初始化

定位、加载、注册，解析成BeanDefinition对象

IOC容器的初始化包括BeanDefinition的Resources定位、载入和注册（工厂模式，根据配置进行生产）这三个基本你的过程。我们以ApplicationContext为例讲解，ApplicationContext系列容器也许是我们最熟悉的，因为web项目中使用XmlWebApplicationContext就属于这个继承体系，还有ClasspathXmlApplicationContext、FileSystemXmlApplicationContext等。

ApplicationContext允许上下文嵌套，通过保存父上下文可以维持一个上下文体系。对于bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享bean定义环境

定位： 得到一个Resource

加载： 得到一个BeanDefinition

注册： register

IOC容器实际上也是一个Map

IOC的存储是根据<bean name=”” id=””> 标签中的name作为key，由IOC容器将Bean包装起来的BeanDefinition作为value

定位和加载的步骤复杂，注册的步骤简单

### 两个ioc容器的创建过程

#### XmlBeanFactory（屌丝IOC）的整个流程

通过XmlBeanFactory的源码，我们可以发现：

public class XmlBeanFactory extends DefaultListableBeanFactory {

private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);

public XmlBeanFactory(Resource resource) throws BeansException {

this(resource, null);

}

public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws BeansException {

super(parentBeanFactory);

this.reader.loadBeanDefinitions(resource);

}

}

调用全过程还原，定位、载入、注册

/\* 根据Xml配置文件创建Resource资源对象，该对象中包含了BeanDefinition的信息 \*/

ClassPathResource resource = new ClassPathResource("application-context.xml");

/\* 创建DefaultListableBeanFactory \*/

DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

/\* 创建XmlBeanDefinitionReader读取器，用于载入BeanDefinition。之所以需要BeanFactory

\* 作为参数，是因为会将读取的信息回调配置给factory

\* \*/

XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);

/\* XmlBeanDefinitionReader执行载入BeanDefinition的方法，最后会完成Bean的载入和注册。

\* 完成后Bean就成功的放置到IOC容器中，以后我们就可以从中取得Bean来使用。

\* \*/

reader.loadBeanDefinitions(resource);

#### 2、FileSystemXmlApplicationContext的IOC容器流程

##### 构造方法：

最终的构造方法：

public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh, ApplicationContext parent) throws BeansException {

super(parent);

//告诉读取器配置文件放在哪里：定位，为了加载配置文件

setConfigLocations(configLocations);

//刷新

if (refresh) {

refresh();

}

}

构造方法实际上做了两个操作：  
首先，调用父类容器的构造方法（super(parent)方法）为容器设置好Bean定义资源文件的定位路径。

通过追踪FileSystemXmlApplicationContext的继承关系，发现其父类的父类AbstractApplicationContext中初始化IOC容器所做的主要源码如下：

static {

//为了避免应用程序在Weblogic8.1关闭时出现类加载异常加载问题，加载IOC容器关

//闭事件（ContextClosedEvent）类

ContextClosedEvent.class.getName();

}

public AbstractApplicationContext() {

this.resourcePatternResolver = getResourcePatternResolver();

}

public AbstractApplicationContext(ApplicationContext parent) {

this();

setParent(parent);

}

//获取一个Spring Source的加载器用于读入Spring Bean定义资源文件

protected ResourcePatternResolver getResourcePatternResolver() {

//AbstractApplicationContext继承DefaultResourceLoader，因此也是一个资源加载器

//Spring资源加载器，其getResource(String location)方法用于载入资源

return new PathMatchingResourcePatternResolver(this);

}

AbstractApplicationContext构造方法中调用PathMatchingResourcePatternResolver的构造方法创建Spring资源加载器：

public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {

Assert.notNull(resourceLoader, "ResourceLoader must not be null");

this.resourceLoader = resourceLoader;

}

在设置容器的资源加载器之后，接下来FileSystemXmlApplicationContext执行setConfigLocations方法通过调用其父类AbstractRefreshalbleConfigApplicationContext的方法进行对Bean定义资源文件的定位，该方法的源码如下：

//解析Bean定义资源文件的路径，处理多个文件字符串数组

String CONFIG\_LOCATION\_DELIMITERS = ",; \t\n";

public void setConfigLocation(String location) { setConfigLocations(StringUtils.tokenizeToStringArray(location,CONFIG\_LOCATION\_DELIMITERS);

}

public void setConfigLocations(String[] locations) {

if (locations != null) {

Assert.noNullElements(locations, "Config locations must not be null");

this.configLocations = new String[locations.length];

for (int i = 0; i < locations.length; i++) {

//resolvePath为同一个类中将字符串解析为路径的方法

this.configLocations[i] = resolvePath(locations[i]).trim();

}

}

else {

this.configLocations = null;

}

}

通过这两个方法的源码我们可以看出，我们即可以使用一个字符串来配置多个Spring Bean定义资源文件，也可以使用字符串数组。

#### 3、AbstractApplicationContext的refresh函数载入Bean定义过程：

Spring IOC容器对Bean定义资源的载入是从refresh()函数开始的，refresh()是一个模板方法，refresh()方法的作用是：在创建IOC容器前，如果已经有容器存在，则需要把已有的容器摧毁和关闭，以保证在refresh之后使用的是新建立起来的IOC容器。refresh的作用类似于对IOC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入。

refresh函数源码：

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

//容器要准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识

prepareRefresh();����

//Bean定义资源文件的载入子类的refreshBeanFactory()方法启动

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

//为BeanFactory配置容器特性，例如加载器、事件处理等

prepareBeanFactory(beanFactory);

try {

//为容器的某些子类指定特殊的BeanPost事件处理器

postProcessBeanFactory(beanFactory);

//调用所有注册的BeanFactoryPostProcessor的Bean

invokeBeanFactoryPostProcessors(beanFactory);

//beanPostProcessor是Bean的后置处理器，用于监听触发的事件

registerBeanPostProcessors(beanFactory);

//初始化信息源，和国际化相关

initMessageSource();

//初始化事件传播器

initApplicationEventMulticaster();

//调用子类的某些特殊Bean初始化

onRefresh();

//为事件传播注册事件监听器

registerListeners();

//初始化所有剩余的单例Bean

finishBeanFactoryInitialization(beanFactory);

//初始化容器的生命周期事件处理器，并发布容器的生命周期事件

finishRefresh();

}

catch (BeansException ex) {

//销毁以创建的单态Bean

destroyBeans();

//取消refresh操作，重置容器的同步标识

cancelRefresh(ex);

throw ex;

}

}

}

refresh()方法主要为IOC容器Bean的生命周期管理提供条件，Spring IOC容器载入Bean定义资源文件从其子类容器的refreshBeanFactory()方法启动，所以整合refresh()中“ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();”这句以后代码的都是注册容器的信息源和生命周期事件，载入过程就是从这句代码启动。

refresh()方法的作用是：在创建IOC容器前，如果已经有容器存在，则需要把已有的容器摧毁和关闭，以保证在refresh之后使用的是新建立起来的IOC容器。refresh的作用类似于对IOC容器的重启，在建立好的容器中对容器进行初始化，对Bean定义资源进行载入。

AbstractApplicationContext的obtainFreshFactory()方法调用子类容器的refreshBeanFactory()方法，启动容器载入Bean定义资源文件的过程，代码如下：

protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {

//这里使用了委派设计模式，父类定义了抽象的refreshBeanFactory()方法，具体实现调 //用子类容器的

refreshBeanFactory();

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

if (logger.isDebugEnabled()) {

logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);

}

return beanFactory;

}

AbstractApplicationContext子类的refreshBeanFactory()方法：

AbstractApplicationContext类中只抽象定义了refreshBeanFactory()方法，容器真正调用的是其子类AbstractRefreshableApplicationContext实现的refreshBeanFactory()方法，方法的源码如下：

@Override

protected final void refreshBeanFactory() throws BeansException {

if (hasBeanFactory()) {//如果已经有容器，摧毁容器中的bean，关闭容器

destroyBeans();

closeBeanFactory();

}

try {

//创建IOC容器

DefaultListableBeanFactory beanFactory = createBeanFactory();

beanFactory.setSerializationId(getId());

//对IOC容器进行定制化，如设置启动参数，开启注解的自动装配

customizeBeanFactory(beanFactory);

//调用载入Bean定义的方法，主要这里有使用了一个委派模式，在当前类中只定

//义了抽象的loadBeanDefinitions方法，具体的实现调用子类容器。

loadBeanDefinitions(beanFactory);

synchronized (this.beanFactoryMonitor) {

this.beanFactory = beanFactory;

}

}

catch (IOException ex) {

throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);

}

}

AbstractRefreshableApplicationContext中指定义了抽象的loadBeanDefinitions方法，容器真正调用的是其子类AbstractXmlApplicationContext对该方法的实现，AbstractXmlApplicationContext的主要源码如下：

loadBeanDefinition方法同样是抽象方法，是由其子类实现的，也即在AbstractXmlApplicationContext中。

@Override

protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {

// 出创建XmlBeanDefinitionReader，即创建Bean容器读取器，并通过回调设置到容器

//中去，容器使用该读取器读取

XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

beanDefinitionReader.setEnvironment(this.getEnvironment());

// 为Bean读取器设置Spring资源加载器，AbstractXmlApplicationContext的祖先父类

//AbstractApplicationContext继承DefaultResourceLoader，因此，容器本身也是一个资源

beanDefinitionReader.setResourceLoader(this);

//为Bean读取器设置SAX xml解析器DOM4J

beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

//当Bean读取器读取Bean定义的Xml资源文件时，启用Xml的校验机制

initBeanDefinitionReader(beanDefinitionReader);

//Bean读取器真实实现加载的方法

loadBeanDefinitions(beanDefinitionReader);

}

protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {

/\*\*

\* 获取Bean定义资源的定位

\* 这里使用了一个委派模式，调用子类的获取Bean定义资源定位的方法

\* 该方法在ClassPathXmlApplicationContext中进行实现，对于我们举例分析源码的

\* FileSystemXmlApplicationContext没有使用方法

\* \*/

Resource[] configResources = getConfigResources();

if (configResources != null) {

//Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的Bean定义 //资源

reader.loadBeanDefinitions(configResources);

}

//如果子类获取的Bean定义资源定位为空，则获取FileSystemXmlApplicationContext构

//造方法中 setContext

String[] configLocations = getConfigLocations();

if (configLocations != null) {

//Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的Bean定义 //资源

reader.loadBeanDefinitions(configLocations);

}

}

Xml Bean读取器（XmlBeanDefinitionReader）调用其父类AbstractBeanDefinitionReader的reader.loadBeanDefinitions方法读取Bean定义资源

由于我们使用FileSystemXmlApplicationContext作为例子分析，因此getConfigResources的返回值为null，因此程序执行reader.loadBeanDefinition（configLocations）分支。

#### 4、AbstractBeanDefinitionReader读取Bean定义资源

AbstractBeanDefinitionReader的loadBeanDefinitions方法源码如下：

可以看到org.springframework.beans.factory.support看一下BeanDefinitionReader的结构，其抽象父类AbstractBeanDefinitionReader中定义了载入过程

public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {

return loadBeanDefinitions(location, null);

}

public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws BeanDefinitionStoreException {

//获取在IOC容器初始化过程中设置的资源加载器

ResourceLoader resourceLoader = getResourceLoader();

if (resourceLoader == null) {

throw new BeanDefinitionStoreException(

"Cannot import bean definitions from location [" + location + "]: no ResourceLoader available");

}

if (resourceLoader instanceof ResourcePatternResolver) {

// Resource pattern matching available.

try {

Resource[] resources

= ((ResourcePatternResolver) resourceLoader).getResources(location);

int loadCount = loadBeanDefinitions(resources);

if (actualResources != null) {

for (Resource resource : resources) {

actualResources.add(resource);

}

}

if (logger.isDebugEnabled()) {

logger.debug("Loaded " + loadCount + " bean definitions from location pattern [" + location + "]");

}

return loadCount;

}

catch (IOException ex) {

throw new BeanDefinitionStoreException(

"Could not resolve bean definition resource pattern [" + location + "]", ex);

}

}

else {

// Can only load single resources by absolute URL.

Resource resource = resourceLoader.getResource(location);

int loadCount = loadBeanDefinitions(resource);

if (actualResources != null) {

actualResources.add(resource);

}

if (logger.isDebugEnabled()) {

logger.debug("Loaded " + loadCount + " bean definitions from location [" + location + "]");

}

return loadCount;

}

}

public int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException {

Assert.notNull(resources, "Resource array must not be null");

int counter = 0;

for (Resource resource : resources) {

counter += loadBeanDefinitions(resource);

}

return counter;

}

loadBeanDefinitions(Resource... resource)方法和上面分析的3个方法类似，同样也是调用XmlBeanDefinitionReader的loadBeanDefinitions方法

从对AbstractBeanDefinitionReader的loadBeanDefinition方法源码分析可以看出该方法做了以下两件事：

首先，调用资源加载器的获取资源方法resourceLoader.getResource(location)，获取到要加载的资源。其次，真正执行加载功能是其子类XmlBeanDefinitionReader的loadBeanDefinitions方法。

<<Interface>>

ResourceLoader

AbstractApplicationContext

getResourcePatternResolver():ResourcePatternResolver

PathMatchingResourcePatternReslvoer

<<Interface>>

ConfigurableApplicationContext

<<Interface>>

ApplicationContext

DefaultResourceLoader

<<Interface>>

ResourcePatternResolver

综合上面的ResourceLoader与ApplicationContext的继承系图，可以知道此时调用的是DefaultResourceLoader中的getSource()方法定位Resource，因为FileSystemXmlApplicationCon

text本身就是DefaultResourceLoader的实现类，所以此时又回到了FileSystemXmlApplication

Context中来

#### 5、资源加载器获取要读入的资源

XmlBeanDefinitionReader通过调用其父类DefaultResourceLoader的getResource方法获取要加载的资源，其源码如下

public Resource getResource(String location) {

Assert.notNull(location, "Location must not be null");

if (location.startsWith(CLASSPATH\_URL\_PREFIX)) {

return new ClassPathResource(location.substring(CLASSPATH\_URL\_PREFIX.length()), getClassLoader());

}

else {

try {

// Try to parse the location as a URL...

URL url = new URL(location);

return new UrlResource(url);

}

catch (MalformedURLException ex) {

// No URL -> resolve as resource path.

return getResourceByPath(location);

}

}

}

FileSystemXmlApplicationContext容器提供了getResourceByPath方法的实现，就是为了处理即不是classpath标识，又不是URL标识的Resource定位这种情况

@Override

protected Resource getResourceByPath(String path) {

if (path != null && path.startsWith("/")) {

path = path.substring(1);

}

return new FileSystemResource(path);

}

这样代码回到了FileSystemXmlApplicationContext中来，它提供了FileSystemResource来完成文件系统得到配置文件的资源定义。

这样，就可以从文件系统路径上对IOC配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在Spring中我们看到它提供的各种资源抽象，比如ClassPathResource，URLResource，FileSystemResource等来供我们使用。上面我们看到的是定位Resource的一个过程，而这只是加载过程的一部分。

#### 6、XmlBeanDefinitionReader加载Bean定义资源：

Bean定义的Resource得到了

继续回到XmlBeanDefinitionReader的loadBeanDefinitions(Resource ...)方法看到代表bean文件的资源定义以后的载入过程。

//XmlBeanDefinitionReader加载资源的入口方法

public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {

//将读入的XML资源进行特殊编码处理

return loadBeanDefinitions(new EncodedResource(resource));

}

//这里是载入XML形式Bean定义资源文件方法

public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {

Assert.notNull(encodedResource, "EncodedResource must not be null");

if (logger.isInfoEnabled()) {

logger.info("Loading XML bean definitions from " + encodedResource.getResource());

}

Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();

if (currentResources == null) {

currentResources = new HashSet<EncodedResource>(4);

this.resourcesCurrentlyBeingLoaded.set(currentResources);

}

if (!currentResources.add(encodedResource)) {

throw new BeanDefinitionStoreException(

"Detected cyclic loading of " + encodedResource + " - check your import definitions!");

}

try {

//将资源文件转为InputStream的IO流

InputStream inputStream = encodedResource.getResource().getInputStream();

try {

//从InputStream中得到Xml的解析源

InputSource inputSource = new InputSource(inputStream);

if (encodedResource.getEncoding() != null) {

inputSource.setEncoding(encodedResource.getEncoding());

}

return doLoadBeanDefinitions(inputSource, encodedResource.getResource());

}

finally {

inputStream.close();

}

}

catch (IOException ex) {

throw new BeanDefinitionStoreException(

"IOException parsing XML document from " + encodedResource.getResource(), ex);

}

finally {

currentResources.remove(encodedResource);

if (currentResources.isEmpty()) {

this.resourcesCurrentlyBeingLoaded.remove();

}

}

}

在Spring框架里面，只要是以do开头的方法，都是具体干活的方法

//从特定的XML文件中实际载入Bean定义资源的方法

protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)

throws BeanDefinitionStoreException {

try {

int validationMode = getValidationModeForResource(resource);

//将XML文件转换为DOM对象，解析过程由documentLoader实现

Document doc = this.documentLoader.loadDocument(

inputSource, getEntityResolver(), this.errorHandler, validationMode, isNamespaceAware());

//这里是启动对Bean定义解析的详细过程，该解析过程会用到Spring的Bean配置 //规则

return registerBeanDefinitions(doc, resource);

} catch (...Exception ex) {

...

}

}

//按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构

public int registerBeanDefinitions(Document doc, Resource resource)

throws BeanDefinitionStoreException {

//得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析

BeanDefinitionDocumentReader documentReader

= createBeanDefinitionDocumentReader();

documentReader.setEnvironment(this.getEnvironment());

//获取容器中注册的Bean数量

int countBefore = getRegistry().getBeanDefinitionCount();

//解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口

//具体的解析实现过程有实现类DefaultBeanDefinitionDocumentRReader完成

documentReader.registerBeanDefinitions(doc, createReaderContext(resource));

//统计解析的Bean数量

return getRegistry().getBeanDefinitionCount() - countBefore;

}

public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {

this.readerContext = readerContext;

logger.debug("Loading bean definitions");

Element root = doc.getDocumentElement();

doRegisterBeanDefinitions(root);

}

#### 7、DocumentLoader将Bean定义资源转为Document对象：

DocumentLoader将Bean定义资源转换成Document对象的源码如下：

//使用标准的JAXP将载入的Bean定义资源装换成document对象

public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,

ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception {

//创建文件解析工厂

DocumentBuilderFactory factory

= createDocumentBuilderFactory(validationMode, namespaceAware);

if (logger.isDebugEnabled()) {

logger.debug("Using JAXP provider [" + factory.getClass().getName() + "]");

}

DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);

return builder.parse(inputSource);

}

//创建文档解析工厂  
protected DocumentBuilderFactory createDocumentBuilderFactory(int validationMode, boolean namespaceAware)throws ParserConfigurationException {

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

factory.setNamespaceAware(namespaceAware);

//设置解析XML的校验

if (validationMode != XmlValidationModeDetector.VALIDATION\_NONE) {

factory.setValidating(true);

if (validationMode == XmlValidationModeDetector.VALIDATION\_XSD) {

// Enforce namespace aware for XSD...

factory.setNamespaceAware(true);

try {

factory.setAttribute(SCHEMA\_LANGUAGE\_ATTRIBUTE, XSD\_SCHEMA\_LANGUAGE);

}

catch (IllegalArgumentException ex) {

ParserConfigurationException pcex = new ParserConfigurationException(

"Unable to validate using XSD: Your JAXP provider [" + factory +

"] does not support XML Schema. Are you running on Java 1.4 with Apache Crimson? " +

"Upgrade to Apache Xerces (or Java 1.5) for full XSD support.");

pcex.initCause(ex);

throw pcex;

}

}

}

return factory;

}

该解析过程调用JavaEE标准的JAXP标准进行处理。

至此Spring IOC容器根据定位的Bean定义资源文件，将其加载读入并转换成为Document对象过程完成。

接下来我们分析Spring IOC容器将载入的Bean定义资源文件转换为Document对象之后，是如何将其解析为Spring IOC管理的Bean对象并将其注册到容器中的。

#### 8、XmlBeanDefinitionReader解析载入的Bean定义资源文件

XmlBeanDefinitionReader类中的doLoadBeanDefinitions方法是从特定XML文件中实际载入Bean定义资源的方法，该方法在载入Bean定义资源之后将其转换为Document对象，接下来调用registerBeanDefinition启动Spring IOC容器对Bean定义的解析过程，registerBeanDefinitions方法源码如下：

//按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构

public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {

//得到BeanDefinitionDocument来对xml格式的BeanDefinition解析

BeanDefinitionDocumentReader documentReader

= createBeanDefinitionDocumentReader();

documentReader.setEnvironment(this.getEnvironment());

//获取容器中注册的Bean数量

int countBefore = getRegistry().getBeanDefinitionCount();

//解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口

//具体的解析实现过程由实现类DefaultBeanDefinitionDocument完成

documentReader.registerBeanDefinitions(doc, createReaderContext(resource));

//统计解析的Bean数量

return getRegistry().getBeanDefinitionCount() - countBefore;

}

//创建BeanDefinitionDocumentReader对象，解析Document对象

protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {

return BeanDefinitionDocumentReader.class.cast(BeanUtils.instantiateClass(this.documentReaderClass));

}

Bean定义资源的载入解析分为以下两个过程：

首先，通过调用Xml解析器将Bean定义资源文件转换得到Document对象，但是这些Document对象并没有按照Spring的Bean规则进行解析。这一步是载入的过程。

其次，在完成通用的XML解析之后，按照Spring的Bean规则对Document对象进行解析。

按照Spring的Bean规则对Document对象解析的的过程是在接口BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader中实现的。

#### 9、DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析：

BeanDefinitionDocumentReader接口通过registerBeanDefinition方法调用其实现类DefaultBeanDefinitionDocumentReader对Document对象进行解析，解析的代码如下：

//根据Spring DTD对Bean的定义规则解析Bean定义Document对象

public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {

//获取Xml描述符

this.readerContext = readerContext;

logger.debug("Loading bean definitions");

//获得Document的根元素

Element root = doc.getDocumentElement();

doRegisterBeanDefinitions(root);

}

protected void doRegisterBeanDefinitions(Element root) {

String profileSpec = root.getAttribute(PROFILE\_ATTRIBUTE);

if (StringUtils.hasText(profileSpec)) {

Assert.state(this.environment != null, "Environment must be set for evaluating profiles");

String[] specifiedProfiles = StringUtils.tokenizeToStringArray(profileSpec,

BeanDefinitionParserDelegate.MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);

if (!this.environment.acceptsProfiles(specifiedProfiles)) {

return;

}

}

BeanDefinitionParserDelegate parent = this.delegate;

//BeanDefinitionParserDelegate中定义了Spring Bean定义XML文件的各种元素

this.delegate = createDelegate(this.readerContext, root, parent);

//在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性

preProcessXml(root);

//从Document的根元素开始进行Bean定义的Document对象

parseBeanDefinitions(root, this.delegate);

//在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性

postProcessXml(root);

this.delegate = parent;

}

//创建BeanDefinitionParserDelegate，用于完成真正的解析过程

protected BeanDefinitionParserDelegate createDelegate(

XmlReaderContext readerContext, Element root, BeanDefinitionParserDelegate parentDelegate) {

BeanDefinitionParserDelegate delegate = createHelper(readerContext, root, parentDelegate);

//BeanDefinitionParserDelegate初始化Document根元素

if (delegate == null) {

delegate = new BeanDefinitionParserDelegate(readerContext, this.environment);

delegate.initDefaults(root, parentDelegate);

}

return delegate;

}

//使用Spring的Bean规则从Document的根元素开始进行Bean定义的Document对象

protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {

//Bean定义的Document对象使用了Spring默认的XML命名空间

if (delegate.isDefaultNamespace(root)) {

//获取Bean定义的Document对象根元素的所有子节点

NodeList nl = root.getChildNodes();

for (int i = 0; i < nl.getLength(); i++) {

Node node = nl.item(i);

//获得Document节点是XML元素节点

if (node instanceof Element) {

Element ele = (Element) node;

//Bean定义的Document的元素节点使用的是Spring默认的XML命名空 //间

if (delegate.isDefaultNamespace(ele)) {

//使用Spring的Bean规则解析元素节点

parseDefaultElement(ele, delegate);

}

else {

//没有使用Spring默认的XML命名空间，则使用用户自定义的解析 //规则解析元素节点

delegate.parseCustomElement(ele);

}

}

}

}

else {

//Document的根节点没有使用Spring默认的命名空间，则使用用户自定义的解析 //规则解析Document根节点

delegate.parseCustomElement(root);

}

}

//使用Spring的Bean规则解析Document元素节点

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {

//如果元素节点是<Import>导入元素，进行导入解析

if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) {

importBeanDefinitionResource(ele);

}

//如果元素节点是<Alias>别名元素，进行别名解析

else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) {

processAliasRegistration(ele);

}

//元素节点既不是导入元素，也不是别名元素，即普通的<Bean>元素，按照Spring的 //Bean规则解析元素

else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) {

processBeanDefinition(ele, delegate);

}

else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) {

// recurse

doRegisterBeanDefinitions(ele);

}

}

##### 如果解析<import>节点就会调用importBeanDefinitionResource(ele)方法：

//解析<Import>导入元素，从给定的导入路径加载Bean定义资源到Spring IoC容器中

protected void importBeanDefinitionResource(Element ele) {

//获取给定的导入元素的location属性

String location = ele.getAttribute(RESOURCE\_ATTRIBUTE);

//如果导入元素的location属性值为空，则没有导入任何资源，直接返回

if (!StringUtils.hasText(location)) {

getReaderContext().error("Resource location must not be empty", ele);

return;

}

// Resolve system properties: e.g. "${user.dir}"

//使用系统变量值解析location属性值

location = environment.resolveRequiredPlaceholders(location);

Set<Resource> actualResources = new LinkedHashSet<Resource>(4);

// Discover whether the location is an absolute or relative URI

//标识给定的导入元素的location是否是绝对路径

boolean absoluteLocation = false;

try {

absoluteLocation = ResourcePatternUtils.isUrl(location) || ResourceUtils.toURI(location).isAbsolute();

}

catch (URISyntaxException ex) {

// cannot convert to an URI, considering the location relative

// unless it is the well-known Spring prefix "classpath\*:"

//给定的导入元素的location不是绝对路径

}

// Absolute or relative?

//给定的导入元素的location是绝对路径

if (absoluteLocation) {

try {

//使用资源读取器加载给定的Bean定义资源

int importCount = getReaderContext().getReader().loadBeanDefinitions(location, actualResources);

if (logger.isDebugEnabled()) {

logger.debug("Imported " + importCount + " bean definitions from URL location [" + location + "]");

}

}

catch (BeanDefinitionStoreException ex) {

getReaderContext().error(

"Failed to import bean definitions from URL location [" + location + "]", ele, ex);

}

}

else {

// No URL -> considering resource location as relative to the current file.

//给定的导入元素的location是相对路径

try {

int importCount;

//将给定导入元素的location是相对路径资源

Resource relativeResource

= getReaderContext().getResource().createRelative(location);

//封装的相对路径资源存在

if (relativeResource.exists()) {

//使用资源读入器加载Bean定义资源

importCount

= getReaderContext().getReader().loadBeanDefinitions(relativeResource);

actualResources.add(relativeResource);

}

//封装的相对路径资源不存在

else {

//获取Spring IOC容器资源读入器的基本路径

String baseLocation = getReaderContext().getResource().getURL().toString();

//根据Spring IOC容器资源读入器的基本路径加载给定导入路径的资源

importCount = getReaderContext().getReader().loadBeanDefinitions(

StringUtils.applyRelativePath(baseLocation, location), actualResources);

}

if (logger.isDebugEnabled()) {

logger.debug("Imported " + importCount + " bean definitions from relative location [" + location + "]");

}

}

catch (IOException ex) {

getReaderContext().error("Failed to resolve current resource location", ele, ex);

}

catch (BeanDefinitionStoreException ex) {

getReaderContext().error("Failed to import bean definitions from relative location [" + location + "]",ele, ex);

}

}

Resource[] actResArray = actualResources.toArray(new Resource[actualResources.size()]);

//在解析完<Import>元素后，发送容器导入其他资源处理完成事件

getReaderContext().fireImportProcessed(location, actResArray, extractSource(ele));

}

##### 如果解析<Alias>节点就会调用processAliasRegistration(ele)方法：

//解析<Alias>别名元素，为Bean向Spring IoC容器注册别名

protected void processAliasRegistration(Element ele) {

//获取<Alias>别名元素中name的属性值

String name = ele.getAttribute(NAME\_ATTRIBUTE);

//获取<Alias>别名元素中alias的属性值

String alias = ele.getAttribute(ALIAS\_ATTRIBUTE);

boolean valid = true;

//<alias>别名元素的name

if (!StringUtils.hasText(name)) {

getReaderContext().error("Name must not be empty", ele);

valid = false;

}

if (!StringUtils.hasText(alias)) {

getReaderContext().error("Alias must not be empty", ele);

valid = false;

}

if (valid) {

try {

//向容器的资源读入器注册别名

getReaderContext().getRegistry().registerAlias(name, alias);

}

catch (Exception ex) {

getReaderContext().error("Failed to register alias '" + alias +

"' for bean with name '" + name + "'", ele, ex);

}

getReaderContext().fireAliasRegistered(name, alias, extractSource(ele));

}

}

##### 如果解析<bean>节点就会调用processBeanDefinition方法：

//解析Bean定义资源Document对象的普通元素

protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {

/\*\*

\* BeanDefinitionHolder是对BeanDefinition的封装，即Bean定义的封装类

\* 对Document对象中<Bean>元素的解析由BeanDefinitionParserDelegate实现

\* BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement

\* \*/

BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);

if (bdHolder != null) {

bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);

try {

// Register the final decorated instance.

//向Spring IOC容器注册解析得到的Bean定义，这是Bean定义向IOC容器注 //册的入口

BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,

getReaderContext().getRegistry());

}

catch (BeanDefinitionStoreException ex) {

getReaderContext().error("Failed to register bean definition with name '" +

bdHolder.getBeanName() + "'", ele, ex);

}

// Send registration event.

//在完成向Spring IOC容器注册解析得到的Bean定义之后，发送注册事件

getReaderContext().fireComponentRegistered(new

BeanComponentDefinition(bdHolder));

}

}

public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {

return parseBeanDefinitionElement(ele, null);

}

public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {

//获取<Bean>元素中的id属性值

String id = ele.getAttribute(ID\_ATTRIBUTE);

//获取<Bean>元素中的name属性值

String nameAttr = ele.getAttribute(NAME\_ATTRIBUTE);

//获取<Bean>元素中的alias属性值

List<String> aliases = new ArrayList<String>();

if (StringUtils.hasLength(nameAttr)) {

String[] nameArr

= StringUtils.tokenizeToStringArray(nameAttr, MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);

aliases.addAll(Arrays.asList(nameArr));

}

String beanName = id;

//如果<Bean>元素中没有配置id属性时，将别名中的第一个值赋值给beanName

if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {

beanName = aliases.remove(0);

if (logger.isDebugEnabled()) {

logger.debug("No XML 'id' specified - using '" + beanName +

"' as bean name and " + aliases + " as aliases");

}

}

//检查<Bean>元素所配置的id或者name的唯一性，containingBean标识<Bean>

//元素中是否包含子<Bean>元素

if (containingBean == null) {

//检查<Bean>元素所配置的id、name或者别名是否重复

checkNameUniqueness(beanName, aliases, ele);

}

//详细对<Bean>元素中配置的Bean定义进行解析的地方

AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);

if (beanDefinition != null) {

if (!StringUtils.hasText(beanName)) {

try {

if (containingBean != null) {

//如果<Bean>元素中没有配置id、别名或者name，且没有包含子元 //素。<Bean>元素，为解析的Bean生产一个唯一beanName注册

beanName = BeanDefinitionReaderUtils.generateBeanName(

beanDefinition, this.readerContext.getRegistry(), true);

}

else {

//如果<Bean>元素中没有配置id、别名或者name，且包含了子元素

//<Bean>元素，为解析的Bean使用别名向IOC容器注册

beanName = this.readerContext.generateBeanName(beanDefinition);

//为解析的Bean使用别名注册时，为了向后兼容

//Spring1.2/2.0，给别名添加类名后缀

String beanClassName = beanDefinition.getBeanClassName();

if (beanClassName != null &&

beanName.startsWith(beanClassName) &&

beanName.length() > beanClassName.length() &&

!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {

aliases.add(beanClassName);

}

}

if (logger.isDebugEnabled()) {

logger.debug("Neither XML 'id' nor 'name' specified - " +

"using generated bean name [" + beanName + "]");

}

}

catch (Exception ex) {

error(ex.getMessage(), ele);

return null;

}

}

String[] aliasesArray = StringUtils.toStringArray(aliases);

return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);

}

return null;

}

通过上述Spring IOC容器对载入Bean定义Document解析可以看出，我们使用Spring时，在Spring配置文件中可以使用<Import>元素来导入IOC容器所需要的其他资源，Spring IoC容器在解析时会首先将指定导入的资源加载进容器中。使用<Alias>别名时，Spring IoC容器首先将别名元素所定义的别名注册到容器中。

对于即不是<Import>元素，又不是<Alias>元素的元素，即Spring配置文件中普通的<Bean>元素解析由BeanDefinitionParserDelegate类的parseBeanDefinitionElement方法来实现。

#### 10、BeanDefinitionParserDelegate解析Bean定义资源文件中的<Bean>元素：

Bean定义资源文件中的<Import>和<Alias>元素解析在DefaultBeanDefinitionDocumentReader中已经完成，对Bean定义资源文件中使用最多的<Bean>元素交由BeanDefinitionParserDelegate来解析，其解析实现的源码如下：

//详细对<Bean>元素中配置的Bean定义其他属性进行解析，由于上面的方法中已经对Bean

//的id、name和别名等属性进行了处理，该方法中主要处理除这三个以外的其他属性数据

public AbstractBeanDefinition parseBeanDefinitionElement(

Element ele, String beanName, BeanDefinition containingBean) {

//记录解析的<Bean>

this.parseState.push(new BeanEntry(beanName));

//这里只读<Bean>元素中配置的class名字，然后载入到BeanDefinition中去

//只是记录配置的class名字，不做实例化在依赖注入时完成

String className = null;

if (ele.hasAttribute(CLASS\_ATTRIBUTE)) {

className = ele.getAttribute(CLASS\_ATTRIBUTE).trim();

}

try {

String parent = null;

//如果<Bean>元素中配置了parent属性，则获取parent属性的值

if (ele.hasAttribute(PARENT\_ATTRIBUTE)) {

parent = ele.getAttribute(PARENT\_ATTRIBUTE);

}

//根据<Bean>元素配置的class名称和parent属性值创建BeanDefinition

//为载入Bean定义信息做准备

AbstractBeanDefinition bd = createBeanDefinition(className, parent);

//对当前的<Bean>元素中配置的一些属性进行解析和设置，如配置的单态（singleton） //属性等

parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);

//为<Bean>元素解析的Bean设置description信息

bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCRIPTION\_ELEMENT));

//对<Bean>元素的meta（元信息）属性解析

parseMetaElements(ele, bd);

//对<Bean>元素的loopup-method属性解析

parseLookupOverrideSubElements(ele, bd.getMethodOverrides());

//对<Bean>元素的replaced-method属性解析

parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

//解析<Bean>元素的构造方法设置

parseConstructorArgElements(ele, bd);

//解析<Bean>元素的<property>设置

parsePropertyElements(ele, bd);

//解析<Bean>元素的qualifier属性

parseQualifierElements(ele, bd);

//为当前解析的Bean设置所需的资源和依赖对象

bd.setResource(this.readerContext.getResource());

bd.setSource(extractSource(ele));

return bd;

}

catch (ClassNotFoundException ex) {

error("Bean class [" + className + "] not found", ele, ex);

}

catch (NoClassDefFoundError err) {

error("Class that bean class [" + className + "] depends on not found", ele, err);

}

catch (Throwable ex) {

error("Unexpected failure during bean definition parsing", ele, ex);

}

finally {

this.parseState.pop();

}

return null;

}

只要使用过Spring，对Spring配置文件比较熟悉的人，通过对上述源码的分析，就会明白我们在Spring配置文件中<Bean>元素中配置的属性就是通过该方法解析和设置到Bean中去的。

注意：在解析<Bean>元素过程中没有创建和实例化Bean对象，只是创建了Bean对象的定义类BeanDefinition，将<Bean>元素中的配置信息设置到BeanDefinition中作为记录，当依赖注入时才使用这些记录信息创建和实例化具体的Bean对象。

上面方法中一些对一些配置如元信息（meta）、qualifier等的解析，我们在Spring中配置时使用的也不多，我们在使用Spring的<Bean>元素时，配置最多的是<Property>属性，因此我们下面继续分析源码，了解Bean的属性在解析时是如何设置的。

#### 11、BeanDefinitionParserDelegate解析<property>元素：

BeanDefinitionParserDelegate在解析<Bean>调用parsePropertyElements方法解析<Bean>元素中的<property>属性子元素，解析源码如下：

//解析<Bean>元素中的<property>子元素

public void parsePropertyElements(Element beanEle, BeanDefinition bd) {

//获取<Bean>元素中所有的子元素

NodeList nl = beanEle.getChildNodes();

for (int i = 0; i < nl.getLength(); i++) {

Node node = nl.item(i);

//如果子元素是<property>子元素，则调用解析<property>子元素方法解析

if (isCandidateElement(node) && nodeNameEquals(node, PROPERTY\_ELEMENT)) {

parsePropertyElement((Element) node, bd);

}

}

}

//解析<property>元素

public void parsePropertyElement(Element ele, BeanDefinition bd) {

//获取<property>元素的名字

String propertyName = ele.getAttribute(NAME\_ATTRIBUTE);

if (!StringUtils.hasLength(propertyName)) {

error("Tag 'property' must have a 'name' attribute", ele);

return;

}

this.parseState.push(new PropertyEntry(propertyName));

try {

//如果一个Bean中已经有同名的property存在，则不进行解析，直接返回。

//即如果在同一个Bean中配置同名的property，则只有第一个起作用

if (bd.getPropertyValues().contains(propertyName)) {

error("Multiple 'property' definitions for property '" + propertyName + "'", ele);

return;

}

//解析获取property的值

Object val = parsePropertyValue(ele, bd, propertyName);

//根据property的名字和值创建property实例

PropertyValue pv = new PropertyValue(propertyName, val);

//解析<property>元素中的属性

parseMetaElements(ele, pv);

pv.setSource(extractSource(ele));

bd.getPropertyValues().addPropertyValue(pv);

}

finally {

this.parseState.pop();

}

}

//解析获取property值

public Object parsePropertyValue(Element ele, BeanDefinition bd, String propertyName) {

String elementName = (propertyName != null) ?

"<property> element for property '" + propertyName + "'" :

"<constructor-arg> element";

// Should only have one child element: ref, value, list, etc.

//获取<property>的所有子元素，只能是其中一种类型：ref，value，list等

NodeList nl = ele.getChildNodes();

Element subElement = null;

for (int i = 0; i < nl.getLength(); i++) {

Node node = nl.item(i);

//子元素不是description和meta属性

if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION\_ELEMENT) &&

!nodeNameEquals(node, META\_ELEMENT)) {

// Child element is what we're looking for.

if (subElement != null) {

error(elementName + " must not contain more than one sub-element", ele);

}

else { //当前<property>元素包括有子元素

subElement = (Element) node;

}

}

}

//判断property的属性值是ref还是value，不允许既是ref又是value

//这个对象封装了ref信息

boolean hasRefAttribute = ele.hasAttribute(REF\_ATTRIBUTE);

boolean hasValueAttribute = ele.hasAttribute(VALUE\_ATTRIBUTE);

if ((hasRefAttribute && hasValueAttribute) ||

((hasRefAttribute || hasValueAttribute) && subElement != null)) {

error(elementName +

" is only allowed to contain either 'ref' attribute OR 'value' attribute OR sub-element", ele);

}

if (hasRefAttribute) {

String refName = ele.getAttribute(REF\_ATTRIBUTE);

if (!StringUtils.hasText(refName)) {

error(elementName + " contains empty 'ref' attribute", ele);

}

//一个指向运行时所依赖对象的引用

RuntimeBeanReference ref = new RuntimeBeanReference(refName);

//设置这个ref的数据对象是被当前的property对象所引用

ref.setSource(extractSource(ele));

return ref;

}

//如果属性时value，创建一个value的数据对象TypedStringValue

//这个对象封装了value信息

else if (hasValueAttribute) {

//一个持有String类型值的对象

TypedStringValue valueHolder

= new TypedStringValue(ele.getAttribute(VALUE\_ATTRIBUTE));

//设置这个value数据对象是被当前的property对象所引用

valueHolder.setSource(extractSource(ele));

return valueHolder;

}

//如果当前<property>元素还有子元素

else if (subElement != null) {

//解析<property>的子元素

return parsePropertySubElement(subElement, bd);

}

else {

// Neither child element nor "ref" or "value" attribute found.

//peoperty的属性既不是ref，也不是value属性，解析出错返回null

error(elementName + " must specify a ref or value", ele);

return null;

}

}

通过对上述源码的分析，我们可以了解在Spring配置文件中，<bean>元素中<property>元素的相关配置时如何处理的：

1. ref被封装为指向依赖对象一个引用。
2. value配置都会封装成一个字符串类型的对象。
3. ref和value都通过“解析的数据类型属性值.setSource(extractSource(ele));”方法将属性值/引用与所引用的属性关联起来。

在方法的最后对于<property>元素的子元素通过parsePropertySubElement方法解析，我们继续分析该方法的源码，了解其解析过程。

#### 12、解析<property>元素得子元素

在beanDefinitionParserDelegeta类中的parsePropertySubElement方法对<property>中的子元素解析，源码如下：

public Object parsePropertySubElement(Element ele, BeanDefinition bd) {

return parsePropertySubElement(ele, bd, null);

}

//解析<property>元素中ref，value或者集合等子元素

public Object parsePropertySubElement(Element ele, BeanDefinition bd, String defaultValueType) {

//如果<property>没有使用Spring默认命名空间，则使用用户自定义的规则解析

//内嵌元素

if (!isDefaultNamespace(ele)) {

return parseNestedCustomElement(ele, bd);

}

//如果子元素是bean，则使用解析<Bean>元素的方法解析

else if (nodeNameEquals(ele, BEAN\_ELEMENT)) {

BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);

if (nestedBd != null) {

nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);

}

return nestedBd;

}

//如果子元素是ref，ref中只有以下3个属性：bean、local、parent

else if (nodeNameEquals(ele, REF\_ELEMENT)) {

// A generic reference to any name of any bean.

//获取<property>元素中的bean属性值，引用其他解析的Bean的名称

//可以不再一个Spring配置文件中，具体请参考Spring对ref的配置规则

String refName = ele.getAttribute(BEAN\_REF\_ATTRIBUTE);

boolean toParent = false;

if (!StringUtils.hasLength(refName)) {

// A reference to the id of another bean in the same XML file.

//获取<property>元素中的local属性值，引用同一个Xml文件中设置的Bean //的id，local和ref不同，local只能引用同一个配置文件中的Bean

refName = ele.getAttribute(LOCAL\_REF\_ATTRIBUTE);

if (!StringUtils.hasLength(refName)) {

// A reference to the id of another bean in a parent context.

//获取<property>元素中parent属性值，引用父级容器中的Bean

refName = ele.getAttribute(PARENT\_REF\_ATTRIBUTE);

toParent = true;

if (!StringUtils.hasLength(refName)) {

error("'bean', 'local' or 'parent' is required for <ref> element", ele);

return null;

}

}

}

//没有配置ref的目标属性值

if (!StringUtils.hasText(refName)) {

error("<ref> element contains empty target attribute", ele);

return null;

}

//创建ref类型数据，指向被引用的对象

RuntimeBeanReference ref = new RuntimeBeanReference(refName, toParent);

//设置引用类型值是被当前子元素所引用

ref.setSource(extractSource(ele));

return ref;

}

//如果子元素<idref>，使用解析ref元素的方法解析

else if (nodeNameEquals(ele, IDREF\_ELEMENT)) {

return parseIdRefElement(ele);

}

//如果子元素<value>，使用解析value元素的方法解析

else if (nodeNameEquals(ele, VALUE\_ELEMENT)) {

return parseValueElement(ele, defaultValueType);

}

//如果子元素是null，为<property>设置一个封装null值的字符串数据

else if (nodeNameEquals(ele, NULL\_ELEMENT)) {

// It's a distinguished null value. Let's wrap it in a TypedStringValue

// object in order to preserve the source location.

TypedStringValue nullHolder = new TypedStringValue(null);

nullHolder.setSource(extractSource(ele));

return nullHolder;

}

//如果子元素是<array>，使用解析array集合子元素的方法解析

else if (nodeNameEquals(ele, ARRAY\_ELEMENT)) {

return parseArrayElement(ele, bd);

}

//如果子元素是<list>，使用解析list结合子元素的方法解析

else if (nodeNameEquals(ele, LIST\_ELEMENT)) {

return parseListElement(ele, bd);

}

//如果子元素是<set>，使用解析set结合子元素的方法解析

else if (nodeNameEquals(ele, SET\_ELEMENT)) {

return parseSetElement(ele, bd);

}

//如果子元素是<map>，使用解析map结合子元素的方法解析

else if (nodeNameEquals(ele, MAP\_ELEMENT)) {

return parseMapElement(ele, bd);

}

//如果子元素是<props>，使用解析props结合子元素的方法解析

else if (nodeNameEquals(ele, PROPS\_ELEMENT)) {

return parsePropsElement(ele);

}

//既不是ref，又不是value，也不是集合，则子元素配置错误，返回null

else {

error("Unknown property sub-element: [" + ele.getNodeName() + "]", ele);

return null;

}

}

通过上述源码分析，我们明白了在Spring配置文件中，对<property>元素中配置Array、List、Set、Map、Prop等各种集合子元素的都通过上述方法解析，生成对应的数据对象，比如ManageList、ManageArray、ManageSet等，这些Manage类是Spring对象BeanDefinition的数据封装，对集合数据类型的具体解析有各自的解析方法实现，解析方法的命名非常规范，一目了然，我们对<list>集合元素的解析方法进行源码分析，了解其实现过程。

12、解析<list>子元素

在BeanDefinitionParserDelegate类中的parseListElement方法就是具体实现解析<property>元素中的<list>集合子元素，源码如下：

//解析<list>集合子元素

public List parseListElement(Element collectionEle, BeanDefinition bd) {

//获取<list>元素中的value-type属性，即获取集合元素的数据类型

String defaultElementType = collectionEle.getAttribute(VALUE\_TYPE\_ATTRIBUTE);

//获取<list>集合元素中的所有子节点

NodeList nl = collectionEle.getChildNodes();

//Spring中将List封装为ManageList

ManagedList<Object> target = new ManagedList<Object>(nl.getLength());

target.setSource(extractSource(collectionEle));

//设置集合目标数据类型

target.setElementTypeName(defaultElementType);

target.setMergeEnabled(parseMergeAttribute(collectionEle));

//具体的<list>元素解析

parseCollectionElements(nl, target, bd, defaultElementType);

return target;

}

//具体解析<list>集合元素，<array>、<list>和<set>都使用该方法解析

protected void parseCollectionElements(

NodeList elementNodes, Collection<Object> target, BeanDefinition bd, String defaultElementType) {

//遍历集合所有节点

for (int i = 0; i < elementNodes.getLength(); i++) {

Node node = elementNodes.item(i);

//节点不是description节点

if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION\_ELEMENT)) {

//将解析的元素加入集合中，递归调用下一个子元素

target.add(parsePropertySubElement((Element) node, bd, defaultElementType));

}

}

}

经过对Spring Bean定义资源文件转换的Document对象中的元素层层解析，Spring IOC现在已经将XML形式定义的Bean定义资源文件转换为Spring IOC所识别的数据结构——BeanDefinition，它是Bean定义资源文件中配置的POJO对象在Spring IOC容器中的映射，我们可以通过AbstractBeanDefinition为入口，看到了IOC容器进行索引、查询和操作。

通过Spring IOC容器对Bean定义资源的解析后，IOC容器大致完成了管理Bean对象的准备工作，即初始化过程，但是最为重要的依赖注入还没有发生，现在在IOC容器中BeanDefinition存储的只是一些静态信息，接下来需要向容器注册Bean定义信息才能全部完成IOC容器的初始化过程

#### 13、解析过后的BeanDefinition在IOC容器中的注册：

让我们继续跟踪程序的执行顺序，接下来会到我们第3步中分析DefaultBeanDefinitionDocumentReader对Bean定义转化的Document对象解析的流程中，在其parseDefaultElement方法中完成对Document对象的解析后得到封装BeanDefinition的BeanDefinitionHold对象，然后调用BeanDefinitionReaderUtils的registerBeanDefinition方法向IOC容器注册解析的Bean，BeanDefinitionReaderUtils的注册的源码如下

//将解析的BeanDefinitionHold注册到容器中

public static void registerBeanDefinition(

BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)

throws BeanDefinitionStoreException {

// Register bean definition under primary name.

//获取解析的BeanDefinition的名称

String beanName = definitionHolder.getBeanName();

//向IOC容器注册BeanDefinition

registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());

// Register aliases for bean name, if any.

//如果解析的BeanDefinition有别名，向容器为其注册别名

String[] aliases = definitionHolder.getAliases();

if (aliases != null) {

for (String aliase : aliases) {

registry.registerAlias(beanName, aliase);

}

}

}

当调用BeanDefinitionReaderUtils向IOC容器注册解析的BeanDefinition时，真正完成注册功能的是DefaultListableBeanFactory。

13、DefaultListableBeanFactory向IOC容器注册解析后的BeanDefinition：

DefaultListableBeanFactory中使用一个HashMap的集合对象存放IOC容器中注册解析的BeanDefinition，向IOC容器注册的主要源码如下：

//存储注册信息的BeanDefinition

private final Map<String, BeanDefinition> beanDefinitionMap

= new ConcurrentHashMap<String, BeanDefinition>(64);

//向IOC容器注册解析的BeanDefinition

public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)

throws BeanDefinitionStoreException {

Assert.hasText(beanName, "Bean name must not be empty");

Assert.notNull(beanDefinition, "BeanDefinition must not be null");

//校验解析的BeanDefinition

if (beanDefinition instanceof AbstractBeanDefinition) {

try {

((AbstractBeanDefinition) beanDefinition).validate();

}

catch (BeanDefinitionValidationException ex) {

throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,

"Validation of bean definition failed", ex);

}

}

//注册的时候需要线程同步，以保证数据的一致性

synchronized (this.beanDefinitionMap) {

Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);

//检查是否有同名的BeanDefinition已经在IOC容器中注册，如果已注册，

//并且不允许覆盖已注册的Bean，则抛出注册失败异常

if (oldBeanDefinition != null) {

if (!this.allowBeanDefinitionOverriding) {

throw new

BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,"Cannot register bean definition [" + beanDefinition + "] for bean '" + beanName +"': There is already [" + oldBeanDefinition + "] bound.");

}

else { //如果允许覆盖，则同名的Bean，后注册的覆盖先注册的

if (this.logger.isInfoEnabled()) {

this.logger.info("Overriding bean definition for bean '" + beanName +

"': replacing [" + oldBeanDefinition + "] with [" + beanDefinition + "]");

}

}

}

//IOC容器中没有已经注册同名的Bean，按正常注册流程注册

else {

this.beanDefinitionNames.add(beanName);

this.frozenBeanDefinitionNames = null;

}

this.beanDefinitionMap.put(beanName, beanDefinition);

}

//重置所有已经注册过的BeanDefinition的缓存

resetBeanDefinition(beanName);

}

至此，Bean定义资源文件中配置的Bean被解析过后，已经注册到IOC容器中，被容器管理起来，真正完成了IOC容器初始化所做的全部工作。现在IOC容器中已经建立了整个Bean的配置信息，这些BeanDefinition信息已经可以使用，并且可以被检索，IOC容器的作用就是对这些注册的Bean定义信息进行处理和维护。这些的注册的Bean定义信息是IOC容器控制反转的基础，正是有了这些注册的数据，容器才可以进行依赖注入。

#### 总结：

现在通过上面的代码，总结一下IOC容器初始化的基本步骤：

1. 初始化的入口在容器实现中的refresh()调用来完成
2. 对bean定义载入IOC容器使用的方法是loadBeanDefinition，

其中的大致过程如下：通过ResourceLoader来完成资源文件位置的定位，DefaultResourceLoader是默认的实现，同时上下文本身就给出了ResourceLoader的实现，可以从类路径，文件系统，URL等方式来定位资源位置。如果是XmlBeanFactory作为IOC容器，那么需要为它指定bean定义资源，也就是说bean定义文件时通过抽象成Resource来被IOC容器处理，容器通过BeanDefinitionReader来完成定义信息的解析和Bean信息的注册，往往使用XmlBeanDefinitionReader来解析bean的xml定义文件-实际的处理过程是委派给BeanDefinitionParserDelegate来完成的，从而得到bean的定义信息，这些信息在Spring中使用BeanDefinition对象来表示-这个名字可以让我们想到loadBeanDefinition，RegisterBeanDefin

ition这些相关方法-他们都是为处理BeanDefinition服务的，容器解析得到BeanDefinitionIOC以后，需要把它在IOC容器中注册，这由IOC实现BeanDefinitionRegistry接口来实现。注册过程就是在IOC容器内部维护一个HashMap来保存得到的BeanDefinition的过程。这个HashMap是IOC容器持有bean信息的场所，以后对bean的操作都是围绕这个HashMap来实现的。

1. 然后我们就可以通过BeanFactory和ApplicationContext来享受到Spring IOC的服务了，在使用IOC容器的时候，我们注意到除了少量粘合代码，绝大多数以正确IOC风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。Spring本身提供了对声明式载入Web应用程序用法的应用程序上下文，并将其存储在ServletContext中的框架实现。

在使用Spring IOC容器的时候我们还需要区别两个概念（写在后面的是主语，写在前面的是修饰语）：

BeanFactory：生成Bean的工厂

FactoryBean：由工厂生产出来的Bean

BeanFactory和FactoryBean，其中BeanFactory指的是IOC容器中的编程抽象，比如ApplicationContext，XmlBeanFactory等，这些都是IOC容器的具体表现，需要使用什么样的容器由客户决定，但Spring为我们提供了丰富的选择。FactoryBean在需要时产生另一个对象，而不返回FactoryBen本身，我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生成的产品。所有的FactoryBean都实现特殊的

org.springframework.beans.factory.FactoryBean接口，当使用容器中FactoryBean的时候，该容器不会返回FactoryBean本身，而是返回其生成的对象。Spring包括了大部分的通用资源和服务访问抽象的FactoryBean的实现，其中包括：对JNDI查询的处理，对代理对象的处理，对事务性代理的处理，对RMI代理的处理等，这些我们都可以看出是具体的工厂，看成是Spring为我们建立好的工厂。也就是说Spring通过使用抽象工厂为我们准备了一系列工厂来生成一些特定的对象，免除我们手工重复的工作，我们要使用时只需要在IOC容器里配置好就能很方便的使用了。

#### IOC重要的部分：

使用了工厂模式（标准化输出商品BeanDefinition）、容器（HashMap）

1. 定位资源（定位查找配置文件）
2. 加载（已经找到的配置文件）
3. 注册（已经加载好的配置文件解析出来并封装成BeanDefinition）

BeanDefinition是对bean的说明而已，bean还没有真正的产生

## 4、IOC容器的依赖注入

### 1、依赖注入发生的时间

当Spring IOC容器完成了Bean定义资源的定位、载入和解析注册以后，IOC容器中已经管理类Bean定义的相关数据，但是IOC容器还没有对所管理的Bean进行依赖注入，依赖注入在以下情况发生：

1. 用户第一次通过getBean方法向IOC容器索要Bean时，IOC容器触发依赖注入
2. 当用户在Bean定义资源中为<Bean>元素配置lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

BeanFactory接口定义了Spring IOC容器的基本功能规范，是Spring IOC容器所应遵守的最底层和最基本的编程规范。BeanFactory接口中定义了几个getBean方法，就是用户向IOC容器索取管理的Bean方法，我们通过分析其子类的具体实现，理解Spring IOC容器在用户索取Bean时如何完成依赖注入。

在BeanFactory中我们看到getBean(String...)函数，它的具体实现在AbstractBeanFactory中

### AbstractBeanFactory的getBean相关方法的源码如下：

//获取IOC容器中指定的Bean

public Object getBean(String name) throws BeansException {

//doGetBean才是真正向IOC容器获取被管理Bean的过程

return doGetBean(name, null, null, false);

}

//获取IOC容器中指定名称和类型的Bean

public <T> T getBean(String name, Class<T> requiredType) throws BeansException {

//doGetBean才是真正向IOC容器获取被管理Bean的过程

return doGetBean(name, requiredType, null, false);

}

//获取IOC容器中指定名称和参数的Bean

public Object getBean(String name, Object... args) throws BeansException {

//doGetBean才是真正向IOC容器获取被管理Bean的过程

return doGetBean(name, null, args, false);

}

//获取IOC容器中指定名称、类型和参数的Bean

public <T> T getBean(String name, Class<T> requiredType, Object... args) throws BeansException {

return doGetBean(name, requiredType, args, false);

}

//真正实现向IOC容器获取Bean的功能，也是触发依赖注入功能的地方

@SuppressWarnings("unchecked")

protected <T> T doGetBean(

final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)

throws BeansException {

//根据指定的名称获取管理Bean的名称。剥离指定名称中对容器的相关依赖

//如果指定的是别名，将别名转换为规范的Bean名称

final String beanName = transformedBeanName(name);

Object bean;

//先从缓存中取是否已经有被创建过的单态类型的Bean

//对于单例模式的Bean整个IOC容器中只创建一次，不需要重复创建

Object sharedInstance = getSingleton(beanName);

//IOC容器创建单例模式Bean实例对象

if (sharedInstance != null && args == null) {

if (logger.isDebugEnabled()) {

//如果指定名称的Bean在容器中已有单例模式的Bean被创建

//直接返回已经创建的Bean

if (isSingletonCurrentlyInCreation(beanName)) {

logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +"' that is not fully initialized yet - a consequence of a circular reference");

} else {

logger.debug("Returning cached instance of singleton bean '" + beanName + "'");

}

}

//获取给定Bean的实例对象，主要是完成FactoryBean的相关处理

//注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是创建对象工厂 //的Bean两者之间由区别

bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);

} else {

//缓存没有正在创建的单例模式Bean

//缓存中已经有已经创建的原型模式Bean

//但是由于循环引用的问题导致实例化对象失败

if (isPrototypeCurrentlyInCreation(beanName)) {

throw new BeanCurrentlyInCreationException(beanName);

}

// 对IOC容器中是否存在名称的BeanDefinition进行检查，首先检查是否能在当前 //的BeanFactory中获取的所需要的Bean，如果不能则委托当前容器的父级容器去 //查找，如果还是找不到则沿着容器的继承体系向父级容器查找

BeanFactory parentBeanFactory = getParentBeanFactory();

//当前容器的父级容器存在，且当前容器不存在指定名称的Bean

if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {

// 解析指定Bean名称的原始名称

String nameToLookup = originalBeanName(name);

if (args != null) {

// 委派父级容器根据指定名称和显示的参数查找

return (T) parentBeanFactory.getBean(nameToLookup, args);

} else {

// 委派父级容器根据指定名称和类型查找

return parentBeanFactory.getBean(nameToLookup, requiredType);

}

}

//创建的Bean是否需要进行类型验证，一般不需要

if (!typeCheckOnly) {

//向容器标记指定的Bean已经被创建

markBeanAsCreated(beanName);

}

try {

//根据指定Bean名称获取其父级的Bean定义

//主要解决Bean继承时子类合并父类公共属性问题

final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

checkMergedBeanDefinition(mbd, beanName, args);

// 获取当前Bean所依赖Bean名称

String[] dependsOn = mbd.getDependsOn();

//如果当前Bean有依赖Bean

if (dependsOn != null) {

for (String dependsOnBean : dependsOn) {

//递归调用getBean方法，获取当前Bean的依赖Bean

getBean(dependsOnBean);

//把被依赖Bean注册给当前依赖的Bean

registerDependentBean(dependsOnBean, beanName);

}

}

// 创建单例模式Bean的实例对象

if (mbd.isSingleton()) {

//这里使用了一个匿名内部类，创建Bean实例对象，并且注册所依赖的 //对象

sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {

public Object getObject() throws BeansException {

try {

return createBean(beanName, mbd, args);

}

catch (BeansException ex) {

//显示的从容器单例模式Bean缓存中清除实例对象

destroySingleton(beanName);

throw ex;

}

}

});

//获取给定Bean的实例对象

bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);

}

//IOC容器创建原型模式Bean实例对象

else if (mbd.isPrototype()) {

// 原型（Prototype）是每次都会创建一个新的对象

Object prototypeInstance = null;

try {

// 回调beforePrototypeCreation方法，默认的功能是注册当前创建的 //原型对象

beforePrototypeCreation(beanName);

//创建指定Bean对象实例

prototypeInstance = createBean(beanName, mbd, args);

}

finally {

//回调afterPrototypeCreate方法，默认功能告诉IOC容器指定Bean //的

afterPrototypeCreation(beanName);

}

//获取Bean的实例对象

bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);

}

//要创建Bean既不是单例模式，也不是原型模式，则根据Bean定义资源中配 //置的生命周期范围，选择实例化Bean的合适方法，这种在Web应用程序中 //比较常用，如request、session、application等生命周期

else {

String scopeName = mbd.getScope();

final Scope scope = this.scopes.get(scopeName);

//Bean定义资源中没有配置生命周期范围，则Bean定义不合法

if (scope == null) {

throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");

}

try {

//这里使用了一个匿名内部类，获取一个指定生命周期范围的实例

Object scopedInstance

= scope.get(beanName, new ObjectFactory<Object>() {

public Object getObject() throws BeansException {

beforePrototypeCreation(beanName);

try {

return createBean(beanName, mbd, args);

}

finally {

afterPrototypeCreation(beanName);

}

}

});

//获取给定Bean实例

bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);

}

catch (IllegalStateException ex) {

throw new BeanCreationException(beanName,

"Scope '" + scopeName + "' is not active for the current thread; " +"consider defining a scoped proxy for this bean if you intend to refer to it from a singleton",ex);

}

}

}

catch (BeansException ex) {

cleanupAfterBeanCreationFailure(beanName);

throw ex;

}

}

// 对创建的Bean实例对象进行类型检查

if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {

try {

return getTypeConverter().convertIfNecessary(bean, requiredType);

}

catch (TypeMismatchException ex) {

if (logger.isDebugEnabled()) {

logger.debug("Failed to convert bean '" + name + "' to required type [" +

ClassUtils.getQualifiedName(requiredType) + "]", ex);

}

throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());

}

}

return (T) bean;

}

doGetBean方法做的操作有:  
1、判断是不是单例（如果有单例，那么就取出来，如果没有就创建一个单例）

2、有相互注入关系的实例化（父类如果是单例直接取出来，如果是一个原型，那么将创建一个）

spring中的对象如果不声明，默认都是单例

通过上面对向IOC容器获取Bean方法的分析，我们可以看到在Spring中，如果Bean定义的单例模式（Singleton），则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式（Prototype），则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

上面只是定义了Bean定义的模式，采取不同创建Bean实例对象的策略，具体的Bean实例对象的创建过程由实现了ObjectFactory接口的匿名内部类的createBean方法完成，Objectfactory使用委派模式，具体的Bean实例创建过程交由其实现类AbstractAutowireCapableBeanFactory完成，我们继续分析AbstractAutowireCapableBeanFactory的createBean方法的源码，理解其创建Bean实例的具体实现过程。

### 3、AbstractAutowireCapableBeanFactory创建Bean实例对象

AbstractAutowireCapableBeanFactory类实现了ObjectFactory接口，创建容器指定的Bean实例对象，同时还对创建的Bean实例对象进行初始化处理。其创建Bean实例对象的方法源码如下：

@Override

//创建Bean实例对象

protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)

throws BeanCreationException {

if (logger.isDebugEnabled()) {

logger.debug("Creating instance of bean '" + beanName + "'");

}

// 判断需要创建的Bean是否可以实例化，即是否可以通过当前的类加载器加载

resolveBeanClass(mbd, beanName);

// 校验和准备Bean中的方法覆盖

try {

mbd.prepareMethodOverrides();

}

catch (BeanDefinitionValidationException ex) {

throw new BeanDefinitionStoreException(mbd.getResourceDescription(),

beanName, "Validation of method overrides failed", ex);

}

try {

// 如果Bean配置了初始化前和初始化后的处理器，则试图返回一个需要创建Bean的代理对象

Object bean = resolveBeforeInstantiation(beanName, mbd);

if (bean != null) {

return bean;

}

}

catch (Throwable ex) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName,

"BeanPostProcessor before instantiation of bean failed", ex);

}

//创建Bean的入口

Object beanInstance = doCreateBean(beanName, mbd, args);

if (logger.isDebugEnabled()) {

logger.debug("Finished creating instance of bean '" + beanName + "'");

}

return beanInstance;

}

//真正创建Bean的方法

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) {

// 封装被创建的Bean对象（BeanWrapper）

BeanWrapper instanceWrapper = null;

if (mbd.isSingleton()) { //单例模式的Bean，先从容器缓存中获取同名Bean

instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

}

if (instanceWrapper == null) {

//创建实例对象

instanceWrapper = createBeanInstance(beanName, mbd, args);

}

final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);

//获取实例化对象的类型

Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);

// 调用PostProcessor

synchronized (mbd.postProcessingLock) {

if (!mbd.postProcessed) {

applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

mbd.postProcessed = true;

}

}

// 向容器中缓存单例模式的Bean对象，以防循环引用

boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&

isSingletonCurrentlyInCreation(beanName));

if (earlySingletonExposure) {

if (logger.isDebugEnabled()) {

logger.debug("Eagerly caching bean '" + beanName +

"' to allow for resolving potential circular references");

}

//这是一个匿名内部类，为了防止循环引用，尽早持有对象的引用

addSingletonFactory(beanName, new ObjectFactory<Object>() {

public Object getObject() throws BeansException {

return getEarlyBeanReference(beanName, mbd, bean);

}

});

}

// Bean对象的初始化，依赖注入在此触发

//这个exposedObject在初始化完成之后返回作为依赖注入的属性值赋值给实例对象

Object exposedObject = bean;

try {

//将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象

populateBean(beanName, mbd, instanceWrapper);

if (exposedObject != null) {

//初始化Bean对象

//在对Bean实例对象生成和依赖注入完成以后，开始对Bean实例对象

//进行初始化，为Bean实例对象应用BeanPostProcessor后置处理器

exposedObject = initializeBean(beanName, exposedObject, mbd);

}

}

catch (Throwable ex) {

if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {

throw (BeanCreationException) ex;

}

else {

throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);

}

}

if (earlySingletonExposure) {

//获取指定名称的已注册的单例模式Bean对象

Object earlySingletonReference = getSingleton(beanName, false);

if (earlySingletonReference != null) {

//根据名称获取的已注册的Bean和正在实例化的Bean是同一个

if (exposedObject == bean) {

//当前实例化的Bean初始化完成

exposedObject = earlySingletonReference;

}

//当前Bean依赖其他Bean，并且当发生循环引用时不允许新创建实例对象

else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {

String[] dependentBeans = getDependentBeans(beanName);

Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);

//获取当前Bean所依赖的其他Bean

for (String dependentBean : dependentBeans) {

//对依赖Bean进行类型检查

if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {

actualDependentBeans.add(dependentBean);

}

}

if (!actualDependentBeans.isEmpty()) {

throw new BeanCurrentlyInCreationException(beanName,

"Bean with name");

}

}

}

}

// 注册完成依赖注入的Bean

try {

registerDisposableBeanIfNecessary(beanName, bean, mbd);

}

catch (BeanDefinitionValidationException ex) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);

}

return exposedObject;

}

通过对方法源码的分析，我们看到具体的依赖注入实现在以下两个方法：

1. createBeanInstance：生成Bean所包含的java对象实例。
2. populateBean：对Bean属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

### 4、createBeanInstance方法创建Bean的java实例对象：

在createBeanInstance方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成java实例对象，创建对象代码如下：

//创建Bean的实例对象

protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {

// 检查确认Bean是可实例化的

Class<?> beanClass = resolveBeanClass(mbd, beanName);

//使用工厂方法对Bean进行实例化

if (beanClass != null

&& !Modifier.isPublic(beanClass.getModifiers())

&& !mbd.isNonPublicAccessAllowed()) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName,

"Bean class isn't public, and non-public access not allowed: " + beanClass.getName());

}

if (mbd.getFactoryMethodName() != null) {

//调用工厂方法实例化

return instantiateUsingFactoryMethod(beanName, mbd, args);

}

// 使用容器的自动装配方法进行实例化

boolean resolved = false;

boolean autowireNecessary = false;

if (args == null) {

synchronized (mbd.constructorArgumentLock) {

if (mbd.resolvedConstructorOrFactoryMethod != null) {

resolved = true;

autowireNecessary = mbd.constructorArgumentsResolved;

}

}

}

if (resolved) {

if (autowireNecessary) {

//配置了自动装配属性，使用容器的自动装配实例化

//容器的自动装配是根据参数类型匹配Bean的构造函数

return autowireConstructor(beanName, mbd, null, null);

}

else {

//使用默认无参构造方法实例化

return instantiateBean(beanName, mbd);

}

}

// 使用Bean的构造方法进行实例化

Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);

if (ctors != null ||

mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR ||

mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {

//使用容器的自动装配特性，调用匹配的构造方法实例化

return autowireConstructor(beanName, mbd, ctors, args);

}

// 使用默认无参构造方法实例化

return instantiateBean(beanName, mbd);

}

//使用默认的无参构造方法实例化Bean对象

protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {

try {

Object beanInstance;

final BeanFactory parent = this;

//获取系统的安全管理接口，JDK标准的安全管理API

if (System.getSecurityManager() != null) {

//这是一个匿名内置类，根据实例化策略创建实例对象

beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {

public Object run() {

return getInstantiationStrategy().instantiate(mbd, beanName, parent);

}

}, getAccessControlContext());

}

else {

//将实例化的对象封装起来

beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);

}

BeanWrapper bw = new BeanWrapperImpl(beanInstance);

initBeanWrapper(bw);

return bw;

}

catch (Throwable ex) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);

}

}

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的Bean的实例化相当比较清楚，调用相应的工厂或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略（JDK的反射机制或者CGLIB）来进行初始化了，在方法getInstantiationStrategy().instantiate中具体实现类使用初始策略实例化对象。

### 5、SimpleInstantiationStrategy类使用默认的无参构造方法创建Bean实例化对象

在使用默认的无参构造方法创建Bean的实例化对象时，方法getInstantiationStrategy().instantiate调用了SimpleInstantiationStrategy类中的实例化Bean的方法，其源码如下：

//使用初始化策略实例化Bean对象

public Object instantiate(RootBeanDefinition beanDefinition, String beanName, BeanFactory owner) {

// 如果Bean定义中没有方法覆盖，则就不需要CGLIB父类类的方法

if (beanDefinition.getMethodOverrides().isEmpty()) {

Constructor<?> constructorToUse;

synchronized (beanDefinition.constructorArgumentLock) {

//获取对象的构造方法或工厂方法

constructorToUse

= (Constructor<?>) beanDefinition.resolvedConstructorOrFactoryMethod;

//如果没有构造方法且没有工厂方法

if (constructorToUse == null) {

//使用JDK的反射机制，判断要实例化的Bean是否是接口

final Class clazz = beanDefinition.getBeanClass();

if (clazz.isInterface()) {

throw new BeanInstantiationException(clazz, "Specified class is an interface");

}

try {

if (System.getSecurityManager() != null) {

//这里是一个匿名内置类，使用反射机制获取Bean的构造方法

constructorToUse = AccessController.doPrivileged(

new PrivilegedExceptionAction<Constructor>() {

public Constructor run() throws Exception {

return clazz.getDeclaredConstructor((Class[]) null);

}

});

}

else {

constructorToUse = clazz.getDeclaredConstructor((Class[]) null);

}

beanDefinition.resolvedConstructorOrFactoryMethod = constructorToUse;

}

catch (Exception ex) {

throw new BeanInstantiationException(clazz, "No default constructor found", ex);

}

}

}

return BeanUtils.instantiateClass(constructorToUse);

}

else {

// 使用CGLIB来实现

return instantiateWithMethodInjection(beanDefinition, beanName, owner);

}

}

通过上面代码分析，我们看到了如果Bean有方法被覆盖了，则使用JDK的反射机制进行实例化，否则，使用CGLIB进行实例化。

instantiateWithMethodInjection方法调用SimpleInstantiationStrategy的子类

CglibSubclassingInstantiationStrategy使用CGLIB来进行初始化，其源码如下：

//使用CGLIB进行Bean对象实例化

public Object instantiate(Constructor ctor, Object[] args) {

//CGLIB中的类

Enhancer enhancer = new Enhancer();

//将Bean本身作为其基类

enhancer.setSuperclass(this.beanDefinition.getBeanClass());

enhancer.setCallbackFilter(new CallbackFilterImpl());

enhancer.setCallbacks(new Callback[] {

NoOp.INSTANCE,

new LookupOverrideMethodInterceptor(),

new ReplaceOverrideMethodInterceptor()

});

//使用CGLIB的create方法生成实例对象

return (ctor == null) ?

enhancer.create() :

enhancer.create(ctor.getParameterTypes(), args);

}

CGLIB是一个常用的字节码生成器的类库，它提供了一系列API实现java字节码的生成和转换功能。我们在学习JDK的动态代理时都知道，JDK的动态代理只能针对接口，如果一个类没有实现任何接口，要对进行动态代理只能使用CGLIB

### 6、populateBean方法对Bean属性的依赖注入：

在第3步的分析中我们已经了解到Bean的依赖注入分为以下两个过程：

（1）createBeanInstance：生成Bean所包含的java对象实例。

（2）populateBean：对Bean属性的依赖注入进行处理。

第4、5步中我们分析了容器初始化生成Bean所包含java实例对象的过程，现在我们继续分析生成对象后，Spring IOC容器是如何将Bean的属性依赖关系注入Bean实例对象中并设置好的，属性依赖注入的代码如下：

//将Bean属性设置到生成的实例对象上

protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {

//获取容器在解析Bean定义资源时为BeanDefinition中设置的属性值

PropertyValues pvs = mbd.getPropertyValues();

//实例对象为null

if (bw == null) {

if (!pvs.isEmpty()) {

throw new BeanCreationException(

mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");

}

else {

// 实例对象为null，属性值为空，不需要设置属性值，直接返回

return;

}

}

//在设置属性之前调用Bean的PostProcessor后置处理器

boolean continueWithPropertyPopulation = true;

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {

continueWithPropertyPopulation = false;

break;

}

}

}

}

if (!continueWithPropertyPopulation) {

return;

}

//依赖注入开始，首先处理autowire自动装配的注入

if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME ||

mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {

MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

// 对autowire自动装配的处理，根据Bean名称自动装配注入

if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME) {

autowireByName(beanName, mbd, bw, newPvs);

}

// 根据Bean类型自动装配注入

if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {

autowireByType(beanName, mbd, bw, newPvs);

}

pvs = newPvs;

}

//检查容器是否持有用于处理单例模式Bean关闭时的后置处理器

boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

//Bean实例对象没有依赖，即没有继承基类

boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY\_CHECK\_NONE);

if (hasInstAwareBpps || needsDepCheck) {

//从实例对象中提取属性描述符

PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);

if (hasInstAwareBpps) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

//使用BeanPostProcessor处理器处理属性值

pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);

if (pvs == null) {

return;

}

}

}

}

if (needsDepCheck) {

//为要设置的属性进行依赖检查

checkDependencies(beanName, mbd, filteredPds, pvs);

}

}

//对属性进行依赖注入

applyPropertyValues(beanName, mbd, bw, pvs);

}

//解析并依赖注入属性的过程

protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {

if (pvs == null || pvs.isEmpty()) {

return;

}

//封装属性值

MutablePropertyValues mpvs = null;

List<PropertyValue> original;

if (System.getSecurityManager()!= null) {

if (bw instanceof BeanWrapperImpl) {

//设置安全上下文，JDK安全机制

((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());

}

}

if (pvs instanceof MutablePropertyValues) {

mpvs = (MutablePropertyValues) pvs;

//属性值已经转换

if (mpvs.isConverted()) {

try {

//为实例化对象设置属性值

bw.setPropertyValues(mpvs);

return;

}

catch (BeansException ex) {

throw new BeanCreationException(

mbd.getResourceDescription(), beanName, "Error setting property values", ex);

}

}

//获取属性值对象的原始类型值

original = mpvs.getPropertyValueList();

} else {

original = Arrays.asList(pvs.getPropertyValues());

}

//获取用户自定义类型的转换

TypeConverter converter = getCustomTypeConverter();

if (converter == null) {

converter = bw;

}

//创建一个Bean定义属性解析器，将Bean定义中的属性值解析为Bean实例对象的实际值

BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);

// 为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中

List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());

boolean resolveNecessary = false;

for (PropertyValue pv : original) {

//属性值不需要转换

if (pv.isConverted()) {

deepCopy.add(pv);

}

//属性值需要转换

else {

String propertyName = pv.getName();

//原始属性值，即转换之前的属性值

Object originalValue = pv.getValue();

//转换属性值，例如将引用转换为IOC容器中实例化对象引用

Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);

//转换之后的属性值

Object convertedValue = resolvedValue;

//属性值是否可以转换

boolean convertible = bw.isWritableProperty(propertyName) &&

!PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);

if (convertible) {

//使用用户自定义的类型转换器转换属性值

convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);

}

// 存储转换后的属性值，避免每次属性注入时的转换工作

if (resolvedValue == originalValue) {

if (convertible) {

//设置属性转换之后的值

pv.setConvertedValue(convertedValue);

}

deepCopy.add(pv);

}

//属性是可转换的，且属性值是字符类型，且属性的原始类型值不是

//动态生成的字符串，且属性的原始值不是集合或者数组类型

else if (convertible && originalValue instanceof TypedStringValue &&

!((TypedStringValue) originalValue).isDynamic() &&

!(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {

pv.setConvertedValue(convertedValue);

deepCopy.add(pv);

}

else {

resolveNecessary = true;

//重新封装属性值

deepCopy.add(new PropertyValue(pv, convertedValue));

}

}

}

if (mpvs != null && !resolveNecessary) {

//标记属性值已经转换过

mpvs.setConverted();

}

// 进行属性依赖注入

try {

bw.setPropertyValues(new MutablePropertyValues(deepCopy));

}

catch (BeansException ex) {

throw new BeanCreationException(

mbd.getResourceDescription(), beanName, "Error setting property values", ex);

}

}

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

1. 属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入
2. 属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在BeanDefinitionValueResolver类中resolveValueIfNecessary方法中进行的，对属性值的依赖注入时通过bw.setPropertyValues方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

### 7、BeanDefinitionValueResolver解析属性值：

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由resolveValueIfNecessary方法实现，其源码如下：

//解析属性值，对注入类型进行转换

public Object resolveValueIfNecessary(Object argName, Object value) {

// 对引用类型的属性进行解析

if (value instanceof RuntimeBeanReference) {

RuntimeBeanReference ref = (RuntimeBeanReference) value;

//调用引用类型属性的解析方法

return resolveReference(argName, ref);

}

//对属性值是引用容器中另一个Bean名称的解析

else if (value instanceof RuntimeBeanNameReference) {

String refName = ((RuntimeBeanNameReference) value).getBeanName();

refName = String.valueOf(evaluate(refName));

//从容器中获取指定名称的Bean

if (!this.beanFactory.containsBean(refName)) {

throw new BeanDefinitionStoreException(

"Invalid bean name '" + refName + "' in bean reference for " + argName);

}

return refName;

}

//对Bean类型属性的解析，主要是Bean中的内部类

else if (value instanceof BeanDefinitionHolder) {

// Resolve BeanDefinitionHolder: contains BeanDefinition with name and aliases.

BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;

return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBeanDefinition());

}

else if (value instanceof BeanDefinition) {

// Resolve plain BeanDefinition, without contained name: use dummy name.

BeanDefinition bd = (BeanDefinition) value;

return resolveInnerBean(argName, "(inner bean)", bd);

}

//对集合数组类型的属性解析

else if (value instanceof ManagedArray) {

// May need to resolve contained runtime references.

ManagedArray array = (ManagedArray) value;

//获取数组的类型

Class<?> elementType = array.resolvedElementType;

if (elementType == null) {

//获取数组元素的类型

String elementTypeName = array.getElementTypeName();

if (StringUtils.hasText(elementTypeName)) {

try {

//使用反射机制创建指定类型的对象

elementType = ClassUtils.forName(elementTypeName, this.beanFactory.getBeanClassLoader());

array.resolvedElementType = elementType;

}

catch (Throwable ex) {

// Improve the message by showing the context.

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Error resolving array type for " + argName, ex);

}

}

//没有获取到数组的类型，也没有获取到数组元素的类型，则直接设置数组的类型为Object

else {

elementType = Object.class;

}

}

return resolveManagedArray(argName, (List<?>) value, elementType);

}

//解析list类型的属性值

else if (value instanceof ManagedList) {

return resolveManagedList(argName, (List<?>) value);

}

//解析set类型的属性值

else if (value instanceof ManagedSet) {

return resolveManagedSet(argName, (Set<?>) value);

}

//解析map类型的属性值

else if (value instanceof ManagedMap) {

return resolveManagedMap(argName, (Map<?, ?>) value);

}

//解析props类型的属性值，props其实就是key和value均为字符串的map

else if (value instanceof ManagedProperties) {

Properties original = (Properties) value;

//创建一个拷贝，用于作为解析后的返回值

Properties copy = new Properties();

for (Map.Entry propEntry : original.entrySet()) {

Object propKey = propEntry.getKey();

Object propValue = propEntry.getValue();

if (propKey instanceof TypedStringValue) {

propKey = evaluate((TypedStringValue) propKey);

}

if (propValue instanceof TypedStringValue) {

propValue = evaluate((TypedStringValue) propValue);

}

copy.put(propKey, propValue);

}

return copy;

}

//解析字符串类型的属性值

else if (value instanceof TypedStringValue) {

TypedStringValue typedStringValue = (TypedStringValue) value;

Object valueObject = evaluate(typedStringValue);

try {

//获取属性的目标类型

Class<?> resolvedTargetType = resolveTargetType(typedStringValue);

if (resolvedTargetType != null) {

return this.typeConverter.convertIfNecessary(valueObject, resolvedTargetType);

}

//没有获取到属性的目标对象，则按Object类型返回

else {

return valueObject;

}

}

catch (Throwable ex) {

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Error converting typed String value for " + argName, ex);

}

}

else {

return evaluate(value);

}

}

//解析引用类型的属性值

private Object resolveReference(Object argName, RuntimeBeanReference ref) {

try {

//获取引用的Bean名称

String refName = ref.getBeanName();

refName = String.valueOf(evaluate(refName));

//如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象

if (ref.isToParent()) {

if (this.beanFactory.getParentBeanFactory() == null) {

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Can't resolve reference to bean '" + refName +

"' in parent factory: no parent factory available");

}

return this.beanFactory.getParentBeanFactory().getBean(refName);

}

//从当前的容器中获取指定的引用对象，如果指定的Bean没有被实例化

//则会递归触发引用Bean的初始化和依赖注入

else {

Object bean = this.beanFactory.getBean(refName);

//将当前实例化对象的依赖引用对象

this.beanFactory.registerDependentBean(refName, this.beanName);

return bean;

}

}

catch (BeansException ex) {

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Cannot resolve reference to bean '" + ref.getBeanName() + "' while setting " + argName, ex);

}

}

//解析array类型的属性

private Object resolveManagedArray(Object argName, List<?> ml, Class<?> elementType) {

//创建一个指定类型的数组，用于存放和返回解析后的数组

Object resolved = Array.newInstance(elementType, ml.size());

for (int i = 0; i < ml.size(); i++) {

//递归解析array的每一个元素，并将解析后的值设置到resolved数组中，索引为i

Array.set(resolved, i,

resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));

}

return resolved;

}

//解析list类型的属性

private List resolveManagedList(Object argName, List<?> ml) {

List<Object> resolved = new ArrayList<Object>(ml.size());

for (int i = 0; i < ml.size(); i++) {

resolved.add(

resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));

}

return resolved;

}

//解析set类型的属性

private Set resolveManagedSet(Object argName, Set<?> ms) {

Set<Object> resolved = new LinkedHashSet<Object>(ms.size());

int i = 0;

//递归解析set的每一个元素

for (Object m : ms) {

resolved.add(resolveValueIfNecessary(new KeyedArgName(argName, i), m));

i++;

}

return resolved;

}

//解析map类型的属性

private Map resolveManagedMap(Object argName, Map<?, ?> mm) {

Map<Object, Object> resolved = new LinkedHashMap<Object, Object>(mm.size());

//递归解析map中每一个元素的key和value

for (Map.Entry entry : mm.entrySet()) {

Object resolvedKey = resolveValueIfNecessary(argName, entry.getKey());

Object resolvedValue = resolveValueIfNecessary(

new KeyedArgName(argName, entry.getKey()), entry.getValue());

resolved.put(resolvedKey, resolvedValue);

}

return resolved;

}

通过上面的代码分析，我们明白了Spring是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是Bean对象实例设置到它所依赖的Bean对象属性上去，在第7步中我们已经说过，依赖注入是通过bw.setPropertyValues方法实现的，该方法也使用了委托模式，在BeanWrapper接口中至少定义了方法声明，依赖注入的具体实现交由其实现类BeanWrapperImpl来完成，下面我们就分析依BeanWrapperImpl中赖注入相关源码。

### BeanWrapperImpl对Bean属性的依赖注入：

BeanWrapperImpl类主要是对容器中完成初始化的Bean实例对象进行属性的依赖注入，即把Bean对象设置它所依赖的另一个Bean的属性中去，依赖注入的相关源码如下：

在AbstractAutowireCapableBeanFactory中有用到bw.setPropertyValues方法：

public void setPropertyValues(PropertyValues pvs) throws BeansException {

setPropertyValues(pvs, false, false);

}

public void setPropertyValues(PropertyValues pvs, boolean ignoreUnknown, boolean ignoreInvalid)

throws BeansException {

......

for (PropertyValue pv : propertyValues) {

try {

setPropertyValue(pv);

}

}

......

}

@Override

public void setPropertyValue(PropertyValue pv) throws BeansException {

PropertyTokenHolder tokens = (PropertyTokenHolder) pv.resolvedTokens;

if (tokens == null) {

......

}

else {

setPropertyValue(tokens, pv);

}

}

@SuppressWarnings("unchecked")

//实现依赖注入功能

private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throws BeansException {

//PropertyTokenHolder主要保存属性的名称、路径，以及集合的size等信息

String propertyName = tokens.canonicalName;

String actualName = tokens.actualName;

//keys是用来保存集合类型属性的size

if (tokens.keys != null) {

// 将属性信息拷贝

PropertyTokenHolder getterTokens = new PropertyTokenHolder();

getterTokens.canonicalName = tokens.canonicalName;

getterTokens.actualName = tokens.actualName;

getterTokens.keys = new String[tokens.keys.length - 1];

System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length - 1);

Object propValue;

try {

//获取属性值，该方法内部使用JDK的内省（Introspector）机制

//调用属性的getter（readerMethod）方法，获取属性的值

propValue = getPropertyValue(getterTokens);

}

catch (NotReadablePropertyException ex) {

throw new NotWritablePropertyException(getRootClass(), this.nestedPath + propertyName,

"Cannot access indexed value in property referenced " +

"in indexed property path '" + propertyName + "'", ex);

}

// 获取集合类型属性的长度

String key = tokens.keys[tokens.keys.length - 1];

if (propValue == null) {

// null map value case

if (this.autoGrowNestedPaths) {

// TODO: cleanup, this is pretty hacky

int lastKeyIndex = tokens.canonicalName.lastIndexOf('[');

getterTokens.canonicalName = tokens.canonicalName.substring(0, lastKeyIndex);

propValue = setDefaultValue(getterTokens);

}

else {

throw new NullValueInNestedPathException(getRootClass(), this.nestedPath + propertyName,

"Cannot access indexed value in property referenced " +

"in indexed property path '" + propertyName + "': returned null");

}

}

//注入array类型的属性值

if (propValue.getClass().isArray()) {

//获取属性的描述符

PropertyDescriptor pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);

//获取数组的类型

Class requiredType = propValue.getClass().getComponentType();

//获取数组的长度

int arrayIndex = Integer.parseInt(key);

Object oldValue = null;

try {

//获取数组以前初始化的值

if (isExtractOldValueForEditor() && arrayIndex < Array.getLength(propValue)) {

oldValue = Array.get(propValue, arrayIndex);

}

//将属性的值赋值给数组中的元素

Object convertedValue = convertIfNecessary(propertyName, oldValue, pv.getValue(),

requiredType, TypeDescriptor.nested(property(pd), tokens.keys.length));

Array.set(propValue, arrayIndex, convertedValue);

}

catch (IndexOutOfBoundsException ex) {

throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,

"Invalid array index in property path '" + propertyName + "'", ex);

}

}

//注入list类型的属性值

else if (propValue instanceof List) {

//获取list集合的类型

PropertyDescriptor pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);

Class requiredType = GenericCollectionTypeResolver.getCollectionReturnType(

pd.getReadMethod(), tokens.keys.length);

List list = (List) propValue;

int index = Integer.parseInt(key);

Object oldValue = null;

if (isExtractOldValueForEditor() && index < list.size()) {

oldValue = list.get(index);

}

//获取list解析后的属性值

Object convertedValue = convertIfNecessary(propertyName, oldValue, pv.getValue(),

requiredType, TypeDescriptor.nested(property(pd), tokens.keys.length));

//获取list集合的size

int size = list.size();

//如果list的长度大于属性值的长度，则多余的元素赋值为null

if (index >= size && index < this.autoGrowCollectionLimit) {

for (int i = size; i < index; i++) {

try {

list.add(null);

}

catch (NullPointerException ex) {

throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,

"Cannot set element with index " + index + " in List of size " +

size + ", accessed using property path '" + propertyName +

"': List does not support filling up gaps with null elements");

}

}

list.add(convertedValue);

}

else {

try {

//为list属性赋值

list.set(index, convertedValue);

}

catch (IndexOutOfBoundsException ex) {

throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,

"Invalid list index in property path '" + propertyName + "'", ex);

}

}

}

//注入map类型的属性值

else if (propValue instanceof Map) {

PropertyDescriptor pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);

//获取map集合key的类型

Class mapKeyType = GenericCollectionTypeResolver.getMapKeyReturnType(

pd.getReadMethod(), tokens.keys.length);

//获取map集合value的类型

Class mapValueType = GenericCollectionTypeResolver.getMapValueReturnType(

pd.getReadMethod(), tokens.keys.length);

Map map = (Map) propValue;

// IMPORTANT: Do not pass full property name in here - property editors

// must not kick in for map keys but rather only for map values.

TypeDescriptor typeDescriptor = (mapKeyType != null ?

TypeDescriptor.valueOf(mapKeyType) : TypeDescriptor.valueOf(Object.class));

Object convertedMapKey = convertIfNecessary(null, null, key, mapKeyType, typeDescriptor);

Object oldValue = null;

if (isExtractOldValueForEditor()) {

oldValue = map.get(convertedMapKey);

}

// Pass full property name and old value in here, since we want full

// conversion ability for map values.

Object convertedMapValue = convertIfNecessary(propertyName, oldValue, pv.getValue(),

mapValueType, TypeDescriptor.nested(property(pd), tokens.keys.length));

map.put(convertedMapKey, convertedMapValue);

}

else {

throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,

"Property referenced in indexed property path '" + propertyName +

"' is neither an array nor a List nor a Map; returned value was [" + pv.getValue() + "]");

}

}

else {

PropertyDescriptor pd = pv.resolvedDescriptor;

if (pd == null || !pd.getWriteMethod().getDeclaringClass().isInstance(this.object)) {

pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);

if (pd == null || pd.getWriteMethod() == null) {

if (pv.isOptional()) {

logger.debug("Ignoring optional value for property '" + actualName +

"' - property not found on bean class [" + getRootClass().getName() + "]");

return;

}

else {

PropertyMatches matches = PropertyMatches.forProperty(propertyName, getRootClass());

throw new NotWritablePropertyException(

getRootClass(), this.nestedPath + propertyName,

matches.buildErrorMessage(), matches.getPossibleMatches());

}

}

pv.getOriginalPropertyValue().resolvedDescriptor = pd;

}

Object oldValue = null;

try {

Object originalValue = pv.getValue();

Object valueToApply = originalValue;

if (!Boolean.FALSE.equals(pv.conversionNecessary)) {

if (pv.isConverted()) {

valueToApply = pv.getConvertedValue();

}

else {

if (isExtractOldValueForEditor() && pd.getReadMethod() != null) {

final Method readMethod = pd.getReadMethod();

if (!Modifier.isPublic(readMethod.getDeclaringClass().getModifiers()) &&

!readMethod.isAccessible()) {

if (System.getSecurityManager()!= null) {

AccessController.doPrivileged(new PrivilegedAction<Object>() {

public Object run() {

readMethod.setAccessible(true);

return null;

}

});

}

else {

readMethod.setAccessible(true);

}

}

try {

if (System.getSecurityManager() != null) {

oldValue = AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {

public Object run() throws Exception {

return readMethod.invoke(object);

}

}, acc);

}

else {

oldValue = readMethod.invoke(object);

}

}

catch (Exception ex) {

if (ex instanceof PrivilegedActionException) {

ex = ((PrivilegedActionException) ex).getException();

}

if (logger.isDebugEnabled()) {

logger.debug("Could not read previous value of property '" +

this.nestedPath + propertyName + "'", ex);

}

}

}

valueToApply = convertForProperty(propertyName, oldValue, originalValue, pd);

}

pv.getOriginalPropertyValue().conversionNecessary = (valueToApply != originalValue);

}

final Method writeMethod = (pd instanceof GenericTypeAwarePropertyDescriptor ?

((GenericTypeAwarePropertyDescriptor) pd).getWriteMethodForActualAccess() :

pd.getWriteMethod());

if (!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers()) && !writeMethod.isAccessible()) {

if (System.getSecurityManager()!= null) {

AccessController.doPrivileged(new PrivilegedAction<Object>() {

public Object run() {

writeMethod.setAccessible(true);

return null;

}

});

}

else {

writeMethod.setAccessible(true);

}

}

final Object value = valueToApply;

if (System.getSecurityManager() != null) {

try {

AccessController.doPrivileged(new PrivilegedExceptionAction<Object>()

public Object run() throws Exception {

writeMethod.invoke(object, value);

return null;

}

}, acc);

}

catch (PrivilegedActionException ex) {

throw ex.getException();

}

}

else {

writeMethod.invoke(this.object, value);

}

}

catch (TypeMismatchException ex) {

throw ex;

}

catch (InvocationTargetException ex) {

PropertyChangeEvent propertyChangeEvent =

new PropertyChangeEvent(this.rootObject, this.nestedPath + propertyName, oldValue, pv.getValue());

if (ex.getTargetException() instanceof ClassCastException) {

throw new TypeMismatchException(propertyChangeEvent, pd.getPropertyType(), ex.getTargetException());

}

else {

throw new MethodInvocationException(propertyChangeEvent, ex.getTargetException());

}

}

catch (Exception ex) {

PropertyChangeEvent pce =

new PropertyChangeEvent(this.rootObject, this.nestedPath + propertyName, oldValue, pv.getValue());

throw new MethodInvocationException(pce, ex);

}

}

}

通过对上面注入依赖代码的分析，我们已经明白了Spring IOC容器是如果将属性的值注入到Bean实例对象中去：

1. 对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性
2. 对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法（reader method）获取指定属性注入以前的值，同时调用属性的setter方法（writer method）为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter注入原理。

至此Spring IOC容器对Bean定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在Spring IOC容器中管理了一系列靠依赖关系联系起来的Bean，程序不需要应用自己手动创建所需的对象，Spring IOC容器会使我们使用的时候自动为我们创建，并且我们追好相关的依赖，这就是Spring核心功能的控制反转和依赖注入的相关功能。

### 总结

1. 获取BeanDefinition信息
2. 调用Factory的createBean方法
   1. createBeanInstance生成Bean所包含的java对象实例，根据情况可能用JDK代理、CGLIB依赖关系有List、Array、Map...
   2. populateBean对Bean属性的依赖注入进行处理。注入，做类型转换

真正的IOC容器，factoryBeanObjectCache

## 5、IOC容器的高级特性

### 1、介绍

通过前面4节对Spring IOC容器的源码分析，我们已经基本上了解了Spring IOC容器对Bean定义资源的定位、读入和解析过程，同时也清除了当用户通过getBean方法向IOC容器获取被管理的Bean时，IOC容器对Bean进行的初始化和依赖注入过程，这些是Spring IOC容器的基本功能特性。Spring IOC容器还有一些高级特性，如果使用lazy-init属性对Bean预初始化、FactoryBean产生或者修饰Bean对象的生成、IOC容器初始化Bean过程中使用BeanPostProcessor后置处理器对Bean生命周期事件管理和IOC容器的autowiring自动装配功能等。

### 2、Spring IOC容器的lazy-init属性实现预实例化：

通过前面我们对IOC容器的实现和工作原理分析，我们知道IOC容器的初始化过程就对Bean定义资源的定位、载入和注册，此时容器对Bean的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取Bean时，通过getBean方法的调用完成。

当Bean定义资源<Bean>元素中配置了lazy-init属性时，容器将会在初始化的时候对所配置的Bean进行预实例化，Bean的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的Bean时，就不再初始化和对Bean进行依赖注入了，直接从容器中获取已经完成依赖注入的现成的Bean，可以提高应用第一次向容器获取Bean的性能。

下面我们通过代码分析容器预实例化的实现过程：

#### （1）refresh()

先从IOC容器的初始化过程开始，通过前面文章分析，我们知道IOC容器读入已经定位的Bean定义资源时从refresh方法开始的，我们首先从AbstractApplicationContext类的refresh方法入手分析，在refresh方法中ConfigurableListableBeanFactory = obtainFreshBeanFactory();启动了Bean定义资源的载入、注册过程，而finishBeanFactoryInitialization方法时对注册后的Bean定义中的预实例化（lazy-init=false，spring默认就是实例化，即为true）的Bean进行处理的地方。

#### （2）finishBeanFactoryInitialization处理预实例化Bean：

当Bean定义资源被载入IOC容器之后，容器将Bean定义资源解析为容器内部的数据结构BeanDefinition注册到容器中，AbstractApplicationContext类中的finishBeanFactoryInitialization方法对配置了预实例化属性的Bean进行预初始化过程

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {

// Initialize conversion service for this context.

if (beanFactory.containsBean(CONVERSION\_SERVICE\_BEAN\_NAME) &&

beanFactory.isTypeMatch(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class)) {

beanFactory.setConversionService(

beanFactory.getBean(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class));

}

// Initialize LoadTimeWeaverAware beans early to allow for registering their transformers early.

String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);

for (String weaverAwareName : weaverAwareNames) {

getBean(weaverAwareName);

}

// Stop using the temporary ClassLoader for type matching.

beanFactory.setTempClassLoader(null);

// Allow for caching all bean definition metadata, not expecting further changes.

beanFactory.freezeConfiguration();

// Instantiate all remaining (non-lazy-init) singletons.

beanFactory.preInstantiateSingletons();

}

ConfigurableListableBeanFactory是一个接口，其preInstantiateSingletons方法由其子类DefaultListableBeanFactory提供。

#### （3）DefaultListableBeanFactory对配置lazy-init属性预实例化：

public void preInstantiateSingletons() throws BeansException {

if (this.logger.isInfoEnabled()) {

this.logger.info("Pre-instantiating singletons in " + this);

}

List<String> beanNames;

synchronized (this.beanDefinitionMap) {

// Iterate over a copy to allow for init methods which in turn register new bean definitions.

// While this may not be part of the regular factory bootstrap, it does otherwise work fine.

beanNames = new ArrayList<String>(this.beanDefinitionNames);

}

for (String beanName : beanNames) {

RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

if (isFactoryBean(beanName)) {

final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY\_BEAN\_PREFIX + beanName);

boolean isEagerInit;

if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {

isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {

public Boolean run() {

return ((SmartFactoryBean<?>) factory).isEagerInit();

}

}, getAccessControlContext());

}

else {

isEagerInit = (factory instanceof SmartFactoryBean &&

((SmartFactoryBean<?>) factory).isEagerInit());

}

if (isEagerInit) {

getBean(beanName);

}

}

else {

getBean(beanName);

}

}

}

}

通过对lazy-init处理源码的分析，我们可以看出，如果设置了lazy-init属性，则容器在完成Bean定义的注册之后，会通过getBean方法，触发对指定Bean的初始化和依赖注入过程，这样应用第一次向容器索取所需的Bean时，容器不在需要对Bean进行初始化和依赖注入，直接从已经完成实例化和依赖注入的Bean中取一个现成的Bean，这样就提高了第一次获取Bean的性能。

### FactoryBean的实现

在Spring中，有两个很容易混淆的类：BeanFactory和FactoryBean。

BeanFactory：Bean工厂，是一个工厂，我们Spring IOC容器最顶层接口就是这个BeanFactory，它的作用是管理Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂Bean，是一个Bean，作用是产生其他bean实例。通常情况下，这种bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他bean实例。通常情况下，bean无须自己实现工厂模式，Spring容器担任工厂角色：但少数情况下，容器中的bean本身就是工厂，其作用是产生其他bean的实例。

当用户使用容器本身时，可以使用转义字符“&”来得到FactoryBean本身，以区别FactoryBean产生的实例对象和FactoryBean对象本身。在BeanFactory中通过如下代码定义了该转义字符：

String FACTORY\_BEAN\_PREFIX = “&”;

如果myJndiObject是一个FactoryBean，则使用&myJndiObject得到的是myJndiObject对象，而不是myJndiObject产生出来的对象：

#### （1）FactoryBean源码如下：

public interface FactoryBean<T> {

T getObject() throws Exception;

Class<?> getObjectType();

boolean isSingleton();

}

#### （2）AbstractBeanFactory的getBean方法调用FactoryBean

在前面我们分析Spring IOC容器实例化Bean并进行依赖注入过程的源码时，提到在getBean方法触发容器实例化Bean的时候会调用AbstractBeanFactory的doGetBean方法来进行实例化的过程，

在上面获取给定的实例对象的getObjectForBeanInstance方法中，会调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean方法，该方法实现了Bean工厂生产Bean实例对象。

Dereference（解引用）：一个在C/C++中应用比较多的术语，在C++中，“\*”是解引用符号，而“&”是引用符号，解引用时指变量指向的是所应用对象的本身数据，而不是引用对象的内存地址。

#### （3）、AbstractBeanFactory生产Bean实例对象：

AbstractBeanFactory类中生产Bean实例对象的主要源码如下：

protected Object getObjectForBeanInstance(

Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {

// Don't let calling code try to dereference the factory if the bean isn't a factory.

if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof FactoryBean)) {

throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());

}

// Now we have the bean instance, which may be a normal bean or a FactoryBean.

// If it's a FactoryBean, we use it to create a bean instance, unless the

// caller actually wants a reference to the factory.

if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {

return beanInstance;

}

Object object = null;

if (mbd == null) {

object = getCachedObjectForFactoryBean(beanName);

}

if (object == null) {

// Return bean instance from factory.

FactoryBean<?> factory = (FactoryBean<?>) beanInstance;

// Caches object obtained from FactoryBean if it is a singleton.

if (mbd == null && containsBeanDefinition(beanName)) {

mbd = getMergedLocalBeanDefinition(beanName);

}

boolean synthetic = (mbd != null && mbd.isSynthetic());

object = getObjectFromFactoryBean(factory, beanName, !synthetic);

}

return object;

}

#### （4）工厂Bean的实现类getObject方法创建Bean实例对象

### 4、BeanPostProcessor后置处理器的实现：

BeanPostProcessor后置处理器是Spring IOC容器经常使用到的一个特性，这个Bean后置处理器是一个监听器，可以监听容器触发的Bean生命周期事件。后置处理器向容器注册以后，容器管理的Bean就具备了接收IOC容器事件回调的能力。

BeanPostProcessor的使用非常简单，值需要一个实现接口BeanPostProcessor的实现类，然后再Bean配置文件中设置即可。

### 5、Spring IOC容器autowiring实现原理

Spring IOC容器提供了两种管理Bean依赖关系的方式：

1. 显式管理：通过BeanDefinition的属性值和构造方法实现Bean依赖关系管理。
2. autowiring：Spring IOC容器的依赖自动装配功能，不需要对Bean属性的依赖关系做显式声明，只需要在配置好autowiring属性，IOC容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自自动匹配容器中管理的Bean，从而自动完成依赖注入。

通过对autowiring自动装配特性的理解，我们知道容器对Bean的自动装配发生在容器对Bean依赖注入的过程中。在前面对Spring IOC容器依赖注入过程源码分析中，我们已经知道了容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory类中的populateBean方法中，我们通过程序分析autowiring的实现原理：

#### （1）AbstractAutoWireCapableBeanFactory对Bean实例进行属性依赖注入：

应用第一次通过getBean方法（配置了lazy-init预实例化属性的除外）向IOC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory的populateBean方法就是实现Bean属性依赖注入的功能，

#### （2）Spring IOC容器根据Bean名称或者类型进行autowiring

#### （3）DefaultSingletonBeanRegistry的registerDependentBean方法对属性注入：

#### 总结：

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：

1. 对Bean的属性代调用getBean方法，完成依赖Bean的初始化和依赖注入。
2. 将依赖Bean的属性引用设置到被依赖的Bean属性上
3. 将依赖Bean的名称和被依赖Bean的名称存储在IOC容器的集合中。

Spring IOC容器的autowiring属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡事都有两面性，自动属性依赖注入也有不足，首先，Bean的依赖关系在配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是Spring容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。